AD-A197 883

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

(4)

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS: DTIC FILE COPY
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: DISTRIBUTION STATEMENT A Approved for public release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): 153-7622	5. MONITORING ORGANIZATION REPORT NUMBER(S): N00014-86-K-0054
6a. NAME OF PERFORMING ORGANIZATION: University of Colorado; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research
6c. ADDRESS (City, State and ZIP Code): Campus Box B-19, Boulder, CO 80309	7b. ADDRESS (City, State and ZIP Code): 800 N. Quincy St., Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS.: PROGRAM ELEMENT NO.; PROJECT NO.; TASK NO.; WORK UNIT
11. TITLE (Include Security Classification): Self-Adaptive Databases	
12. PERSONAL AUTHOR(S): Roger King	
13a. TYPE OF REPORT: Final; 13b. TIME COVERED: FROM 9/1/85 TO 7/31/88	14. DATE OF REPORT (Yr., Mo., Day): 88/07/31; 15. PAGE COUNT: 135
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES: FIELD; GROUP; SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

AUG 16 1988

S D

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The self-adaptive databases project at the University of Colorado has produced several substantial results. Parallel algorithms for the maintenance of derived data in an object-oriented database management system have been developed. These algorithms dramatically reduce the amount of I/O necessary to keep complex engineering database entities up to date. Mechanisms have been developed which integrate two directions which have been prominent in the database research community - behavioral and structural (or "semantic") object-oriented modeling. This has allowed the support of data objects which are both structurally complex and behaviorally powerful. This is crucial in supporting emerging engineering applications. Also, the project has resulted in the development of mechanisms for the self-adaptive clustering of data and the self-adaptive scheduling of database updates according to usage patterns. A self-adaptive approach is seen as a promising way to solve the well-known short-coming of relational databases - they are typically too slow to provide proper support of engineering systems.

Keywords:

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code); 22c. OFFICE SYMBOL: Management Systems (R)

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Final Report on ONR N00014-86-K-0054

Roger King

University of Colorado
Department of Computer Science
Boulder, Colorado 80309

The self-adaptive databases project at the University of Colorado has produced several substantial results. Parallel algorithms for the maintenance of derived data in an object-oriented database management system have been developed. These algorithms dramatically reduce the amount of I/O necessary to keep complex engineering database entities up to date.

Mechanisms have been developed which integrate two directions which have been prominent in the database research community - behavioral and structural (or "semantic") object-oriented modeling. This has allowed the support of data objects which are both structurally complex and behaviorally powerful. This is crucial in supporting emerging engineering applications.

Also, the project has resulted in the development of mechanisms for the self-adaptive clustering of data and the self-adaptive scheduling of database updates according to usage patterns. A self-adaptive approach is seen as a promising way to solve the well-known short-coming of relational databases - they are typically too slow to provide proper support of engineering systems.

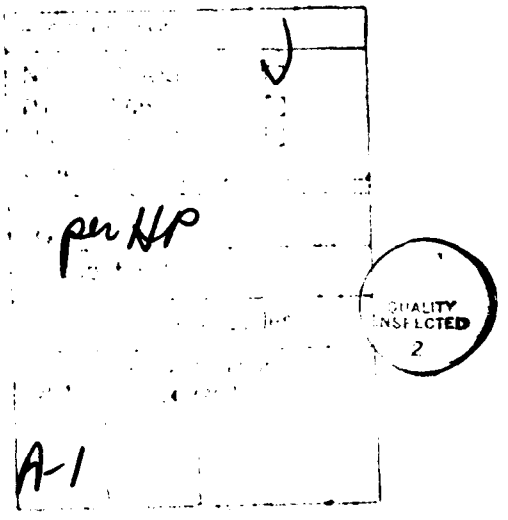
Finally, a prototype system has been implemented, in order to provide a basis for experimentation and for the evolution of the underlying algorithms. In particular, substantial experiments have been performed in order to illustrate that the techniques developed are useful for engineering databases.

The algorithms and self-adaptive mechanisms, as well as the prototype implementation are described in detail in [2, 5], and the application of this system to engineering databases is discussed in [1, 3, 4]. Semantic models are described in [6].

All of the papers referenced in this report were derived from work supported by this project. The three journal papers which are referenced are included with this report, in order to provide more details on the result of the project. Work is continuing, with the parallel algorithms being adapted and expanded to handle the distribution of complex, derived data over a network of databases. As engineering design applications are naturally distributed, this is seen as an important research direction.

References

1. J. Bein, P. Drew and R. King, "Experience with Two Database Tools to Support Software Engineering", *University of Colorado Technical Report*, 1988.
2. S. Hudson and R. King, "CACTIS: A Database System for Specifying Functionally-Defined Data", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 26-37.
3. S. E. Hudson and R. King, "Object-oriented database support for software environments", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, California, May, 1987, 491-503.
4. S. Hudson and R. King, "The Cactis Project: Database Support for Software Engineering", *IEEE Trans. on Software Engineering*, June 1988.
5. S. Hudson and R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM Transactions on Database Systems*, to appear.
6. R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues", *ACM Computing Surveys*, September 1987.



Semantic Database Modeling: Survey, Applications, and Research Issues

RICHARD HULL

Computer Science Department, University of Southern California, Los Angeles, California 90089-0782

ROGER KING

Computer Science Department, University of Colorado, Boulder, Colorado 80309

Most common database management systems represent information in a simple record-based format. Semantic modeling provides richer data structuring capabilities for database applications. In particular, research in this area has articulated a number of constructs that provide mechanisms for representing structurally complex interrelations among data typically arising in commercial applications. In general terms, semantic modeling complements work on knowledge representation (in artificial intelligence) and on the new generation of database models based on the object-oriented paradigm of programming languages.

This paper presents an in-depth discussion of semantic data modeling. It reviews the philosophical motivations of semantic models, including the need for high-level modeling abstractions and the reduction of semantic overloading of data type constructors. It then provides a tutorial introduction to the primary components of semantic models, which are the explicit representation of objects, attributes of and relationships among objects, type constructors for building complex types, ISA relationships, and derived schema components. Next, a survey of the prominent semantic models in the literature is presented. Further, since a broad area of research has developed around semantic modeling, a number of related topics based on these models are discussed, including data languages, graphical interfaces, theoretical investigations, and physical implementation strategies.

Categories and Subject Descriptors: H.0 [**Information Systems**] General; H.2.1 [**Database Management**] Logical Design—*data models*; H.2.2 [**Database Management**] Physical Design—*access methods*; H.2.3 [**Database Management**] Languages—*data description languages (DDL); data manipulation languages (DML); query languages*

General Terms: Design, Languages

Additional Key Words and Phrases: Conceptual database design, entity-relationship model, functional data model, knowledge representation, semantic database model

INTRODUCTION

Commercial database management systems have been available for two decades, originally in the form of the hierarchical and network models. Two opposing research directions in databases were initiated in the early 1970s, namely, the introduction of the relational model and the development of semantic database models. The relational model revolutionized the field by separating logical data representation from physical implementation. Significantly, the inherent simplicity in the model permitted the development of powerful, nonprocedural query languages and a variety of useful theoretical results.

The history of semantic modeling research is quite different. Semantic models were introduced primarily as schema design tools: A schema could first be designed in a high-level semantic model and then translated into one of the traditional models for ultimate implementation. The emphasis of the initial semantic models was to accurately model data relationships that arise frequently in typical database applications. Consequently, semantic models are more complex than the relational model and encourage a more navigational view of data relationships. The field of semantic models is continuing to evolve. There has been increasing interest in using these models as the bases for full-fledged database management systems or at least as complete front ends to existing systems.

The first published semantic model appeared in 1974 [Abriel 1974]. The area matured during the subsequent decade, with the development of several prominent models and a large body of related research efforts. The central result of semantic modeling research has been the development of powerful mechanisms for representing the structural aspects of business data. In recent years, database researchers have turned their attention toward incorporating the behavioral (or dynamic) aspects of data into modeling formalisms; this work is being heavily influenced by the object-oriented paradigm from programming languages.

This paper provides both a survey and a tutorial on semantic modeling and related research. In keeping with the historical emphasis of the field, the primary focus is on the structural aspects of semantic models; a secondary emphasis is given to their behavioral aspects. We begin by giving a broad overview of the fundamental components and the philosophical roots of semantic modeling (Section 1). We also discuss the relationship of semantic modeling to other research areas of computer science. In particular, we discuss important differences between the constructs found in semantic models and in object-oriented programming languages. In Section 2 we use a Generic Semantic Model to provide a detailed, comprehensive tutorial that describes, compares, and contrasts the various semantic constructs found in the literature. In Section 3, we survey a number of published models. We conclude with an overview of ongoing research directions that have grown out of semantic modeling (Section 4); these include database systems and graphical interfaces based on semantic models and theoretical investigations of semantic modeling.

Semantic data models and related issues are described in the earlier survey article by Kerschberg et al. [1976] by Tsichritzis and Lochovsky [1982], and the collection of articles that comprise Brodie et al. [1984]. Also, Afsarmanesh and McLeod [1984], King and McLeod [1985b], and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1988 ACM 0360-0300/87/0900-0201 $1.50

CONTENTS

Maryanski and Peckham [1986] present taxonomies of the more prominent models, and Urban and Delcambre [1986] survey several semantic models, with an emphasis on features in support of temporal information. The dynamic aspects of semantic modeling are emphasized in Borgida [1985]. The overall focus of the present paper is somewhat different from these other surveys in that here we discuss both the prominent semantic models and the research directions they have spawned.

1. PHILOSOPHICAL CONSIDERATIONS

There is an analogy between the motivations behind semantic models and those behind high-level programming languages. The ALGOL-like languages were developed in an attempt to provide richer, more convenient programming abstractions; they buffer the user from low-level machine considerations. Similarly, semantic models attempt to provide more powerful abstractions for the specification of database schemas than are supported by the relational, hierarchical, and network models. Of course, more complex abstraction mechanisms introduce implementation issues. The construction of efficient semantic databases is an interesting problem—and largely an open research area.

In this section we focus on the major motivations and advantages of semantic database modeling as described in the literature. These were originally proposed in, for example, Hammer and McLeod [1981], Kent [1978], Kent [1979], and Smith and Smith [1977] and have since been echoed and extended in works such as Abiteboul and Hull [1987], Brodie [1984], King and McLeod [1985b], and Tsichritzis and Lochovsky [1982].

Historically, semantic database models were first developed to facilitate the design of database schemas (Chen 1976; Hammer and McLeod 1981; Smith and Smith 1977). In the 1970s, the traditional models (relational, hierarchical, and network) were gaining wide acceptance as efficient data management tools. The data structures used in these models are relatively close to those used for the physical representation of data in computers, ultimately viewing data as collections of records with printable or pointer field values. Indeed, these models are often referred to as being *record based*. Semantic models were developed to provide a higher level of abstraction for modeling data, allowing database designers to think of data in ways that correlate more directly to how data arise in the world. Unlike the traditional models, the constructs of most semantic models naturally support a top-down, modular view of the schema, thus simplifying both schema design and database usage. Indeed, although the semantic models were first introduced as design tools, there is increasing interest and research directed toward developing them into full-fledged database management systems.

To present the philosophy and advantages of semantic database models in more detail, we begin by introducing a simple example using a generic semantic data model, along with a corresponding third normal form (3NF) relational schema. The example is used for several purposes. First, we present the fundamental differences between semantic models and the object-oriented paradigm from programming languages. Next, we illustrate the primary advantages often cited in the literature of semantic data models over the record-oriented models. We then show how these advantages relate to the process of schema design. We conclude by comparing semantic models with the related field of knowledge representation in AI.

1.1 An Example

The sample schema shown in Figure 1 is used to provide an informal introduction to many of the fundamental components of semantic data models. This schema is based on a generic model, called the Generic Semantic Model (GSM), which was developed for this survey and is presented in detail in Section 2.

The primary components of semantic models are the explicit representation of objects, attributes of and relationships among objects, type constructors for building complex types, ISA relationships, and

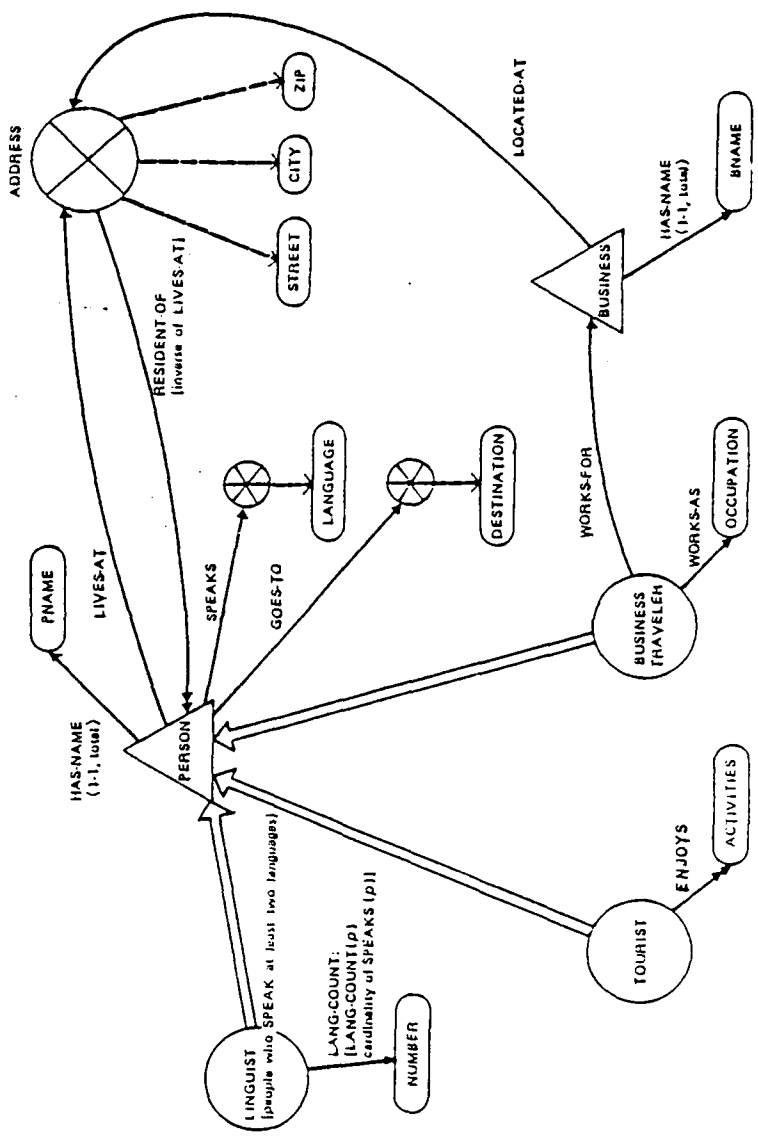


Figure 1. Schema of World Traveler database.

derived schema components. The example schema provides a brief introduction to each of these. The schema corresponds to a mythical database, called the World Traveler Database, which contains information about both business and pleasure travelers. It is necessarily simplistic but highlights the primary features common to the prominent semantic database models.

The World Traveler schema represents two fundamental object or entity types, corresponding to the types PERSON and BUSINESS. These are depicted using triangle nodes, indicating that they correspond to *abstract* data types in the world. Speaking conceptually, in an *instance* of this schema, a set of objects of type PERSON is associated with the PERSON node. In typical implementations of semantic data models [Atkinson and Kulkarni 1983; King 1984; Smith et al. 1981] (see Section 4.1), these abstract objects are referenced using internal identifiers that are not visible to the user. A primary reason for this is that objects in a semantic data model may not be uniquely identifiable using printable attributes that are directly associated with them. In contrast with abstract types, *printable* types such as PNAME (personname) are depicted using ovals. (In the work by Verheijen and Bekkum [1982], which considers the design of information systems, printable types are called lexical object types (LOT) and abstract types are called nonlexical object types (NOLOT).

The schema also represents three subtypes of the type PERSON, namely, TOURIST, BUSINESS-TRAVELER, and LINGUIST. Such subtype/supertype relationships are also called *ISA* relationships; for example, each tourist "is-a" person. In the schema, the three subtypes are depicted using circular nodes (indicating that their underlying type is given elsewhere in the schema), along with double-shafted ISA arrows indicating the ISA relationships. In an instance of this schema, subsets of the set of persons (i.e., the set of internal identifiers associated with PERSON node) would be associated with each of the three subtype nodes. Note that in the absence of any restrictions, the sets corresponding to these subtypes may overlap.

The sample schema illustrates two fundamental uses of subtyping in semantic models, these being to form *user-specified* and *derived* subtypes. For example, the subtypes TOURIST and BUSINESS-TRAVELER are viewed here as being user specified because a person will take on either (or both) of these roles only if this is specified by a database operation. In contrast, we assume here (again simplistically) that a person is a LINGUIST if that person can speak at least two languages. (The attribute SPEAKS that is defined on PERSON is discussed shortly.) Thus, the contents of the subtype LINGUIST can be *derived* from data stored elsewhere in the schema, along with the defining predicate (in pseudo-English) "LINGUIST := PERSONs who SPEAK at least two LANGUAGEs". This example illustrates one type of derived schema component typical of semantic models.

The sample schema also illustrates how *constructed types* can be built from atomic types in a semantic data model. One example of a constructed type is ADDRESS, which is an *aggregation* (i.e., Cartesian product) of three printable types STREET, CITY, and ZIP. This is depicted in the schema with an ×-node that has three children corresponding to the three coordinates of the aggregation. Aggregation is one form of *abstraction* offered by most semantic data models. For example, here it allows users to focus on the abstract notion of ADDRESS while ignoring its component parts. As we shall see, this aggregate object will be referenced by two different parts of the schema. A second prominent type constructor in many semantic models is called *grouping*, or *association* (i.e., finitary powerset) and is used to build sets of elements of an existing type. In the schema, grouping is depicted by a •-node and is used to form, for example, sets of LANGUAGEs and DESTINATIONs.

As illustrated above, object types can be modeled in a semantic schema as being abstract, printable, or constructed and can be defined using an ISA relationship. Through this flexibility the schema designer may choose a construct appropriate to the significance of the object type in the

particular application environment. For example, in a situation in which cities play a more prominent role (e.g., if CITY had associated attributes such as language or climate information), the type of city could be modeled as an abstract type instead of as a printable. As discussed below, different combinations of other semantic modeling constructs provide further flexibility.

So far, we have focused on how object types and subtypes can be represented in semantic data models. Another fundamental component of most semantic models consists of mechanisms for representing *attributes* (i.e., functions) associated with these types and subtypes. It should be noted that unlike the functions typically found in programming languages, many attributes arising in semantic database schemas are not computed but instead are specified explicitly by the user to correspond to facts in the world. In the World Traveler Database, attributes are represented using (single-shafted) arrows originating at the domain of the attribute and terminating at its range. For example, the type PERSON has four attributes: HAS-NAME, which maps to the printable type PNAME; LIVES-AT, which maps to objects of type ADDRESS; SPEAKS, which maps each person to the set of languages that person speaks; and GOES-TO, which maps each person to the set of destinations that person frequents. In the schema the HAS-NAME attribute is constrained to be a 1:1, total function. The attribute SPEAKS is set valued in the sense that the attribute associates a *set* of languages (indicated by the •-node) to each person. RESIDENT-OF is similar in that it associates a set of people with an address; however, this property is represented with a *multivalued* attribute. ENJOYS of TOURIST is also multivalued. The distinction between set valued and multivalued attributes is discussed in Section 2. In several models it is typical to depict both an attribute and its inverse. For example, in the sample schema, the inverse of the LIVES-AT attribute from PERSON to ADDRESS is a set-valued attribute RESIDENT-OF.

As shown in the schema, the subtype BUSINESS-TRAVELER has two attributes: WORKS-FOR and WORKS-AS. Because business travelers are people, the members of this subtype also *inherit* the four attributes of the type PERSON. Similarly, the other two subtypes of PERSON inherit these attributes of type PERSON.

The schema also illustrates how attributes can serve as derived schema components. One example is the attribute RESIDENT-OF; another is the attribute LANG-COUNT of the (derived) subtype LINGUIST, which is specified completely by the predicate "LANG-COUNT is cardinality of SPEAKS" and other parts of the schema.

To conclude this section, Figure 2 shows a 3NF [Ullman 1982] relational schema corresponding to the World Traveler schema. In order to capture most of the semantics of the original schema, key and inclusion dependencies are included in the relational schema. (Briefly, a *key dependency* states that the value of one (or several) field(s) of a tuple determines the remaining field values of that tuple; an *inclusion dependency* states that all of the values occurring in one (or more) column(s) of one relation also occur in some column(s) of another relation.) For example, PNAME is the key of PERSON, indicating that each person has only one address; and the PNAME column of TOURIST is contained in the PNAME column of PERSON, indicating that each tourist is a person. In this schema one or more relations is used for each of the object types in the semantic schema. For example, even ignoring the subtypes of the type PERSON, information about persons is stored in the three relations PERSON, PERSPEAKS, and PERGOES. (In principle, a single relation could be used for this information, but in the presence of set-valued attributes such as SPEAKS and GOES-TO, such relations will not be in 3NF.)

1.2 Semantic Models versus Object-Oriented Programming Languages

Now that we have briefly introduced the essentials of semantic modeling, we are in a position to describe the fundamental distinctions between semantic models and

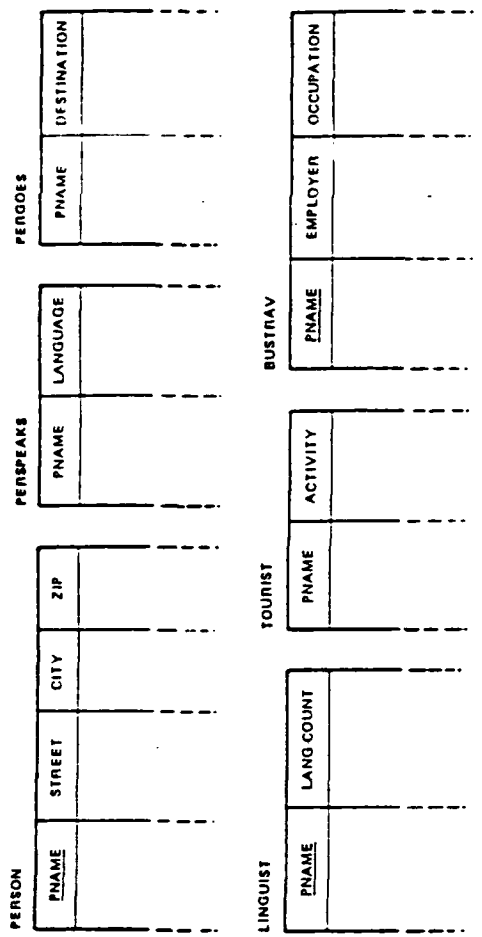


Figure 2. 3NF relational schema corresponding to the World Traveler schema. (a) Relations. (b) Inclusion dependencies.

object-oriented programming [Bobrow et al. 1986; Goldberg and Robson 1983; Moon 1986]. This is crucial in light of current database research thrusts.

Essentially, semantic models encapsulate structural aspects of objects, whereas object-oriented languages encapsulate behavioral aspects of objects. Historically, object-oriented languages stem from research on abstract data types [Guttag 1977; Liskov et al. 1977]. There are three principle features of object-oriented languages. The first is the explicit representation of object classes (or types). Objects are identified by surrogates rather than by their values. The second feature is the encapsulation of "methods" or operations within objects. For example, the object type GEOMETRIC_OBJECT may have the method "display_self". Users are free to ignore the implementation details of methods. The final feature of object-oriented languages is the inheritance of methods from one class to another.

There are two central distinctions between this approach and that of semantic models. First, object-oriented models do not typically embody the rich type constructors of semantic models. From the structural point of view, object-oriented models support only the ability to define single- and multivalued attributes. Second, the inheritance of methods is strictly different from the inheritance of attributes (as in semantic models). In a semantic model, the inheritance of attributes is only between types where one is a subset of the other. The inheritance of a method, since it is a behavioral—and not a structural—property, can be between seemingly unlike types. Thus, the object type TEXT might be able to inherit the "display_self" method of GEOMETRIC_OBJECT.

1.3 Advantages of Semantic Data Models

In this section we summarize the motivations often cited in the literature in support of semantic data models over the traditional data models. We noted above that semantic data models were first introduced primarily as schema design tools and embody the fundamental kinds of relationships arising in typical database applications. As a result of this philosphical foundation, semantically based data models and systems provide the following advantages over traditional, record-oriented systems:

(1) increased separation of conceptual and physical components,
(2) decreased semantic overloading of relationship types,
(3) availab lity of convenient abstraction mechanisms.

Abstraction mechanisms are the means by which the first two advantages of semantic models are obtained. We discuss abstraction separately because of the significant effort researchers have put into developing these mechanisms. Each of the three advantages is discussed below.

1.3.1 Increased Separation of Logical and Physical Components

In record-oriented models the access paths available to end users tend to mimic the logical structure of the database schema directly [Chen 1976; Hammer and McLeod 1981; Kent 1979; Kerschberg and Pacheco 1979; Shipman 1981; Smith and Smith 1977]. This phenomenom exhibits itself in different ways in the relational and the hierarchical/network models. In the relational model a user must simulate pointers by comparing identifiers in order to traverse from one relation to another (typically using the join operator). In contrast, the attributes of semantic models may be used as direct conceptual pointers. Thus, users must consciously traverse through an extra level of indirection imposed by the relational model, making it more difficult to form complex objects out of simpler ones. For this reason, the relational model has been referred to as being *value oriented* [Khoshafian and Copeland 1986; Ullman 1987] as opposed to *object oriented*.

In the hierarchical and network models a similar situation occurs. Users must navigate through the database, constructing larger objects out of flat record structures by associating records of different types. In contrast, semantic models allow users to

focus their attention directly on abstract objects. Thus, in a hierarchical/network model, the access paths correspond directly to the low-level physical links between records and not to the conceptual relationships modeled in a semantic schema.

To illustrate this point using the relational model, suppose that in the World Traveler database Mary is a business traveler. Using attributes, the city of Mary's employer can be obtained with the simple query:

```
print LOCATED-AT (WORKS-
  FOR('Mary')).CITY
```

This query operates as follows: Mary's employer is obtained by WORKS-FOR('Mary'); applying LOCATED-AT yields the address of that employer, and the '.CITY' construct isolates the second coordinate of the address. (We assume as syntactic sugar that because HAS-NAME is 1:1, the string 'Mary' can be used to denote the person Mary; if not, in the above query, 'Mary' would have to be replaced by HAS-NAME^{-1}('Mary').) Thus, the semantic model permits users to refer to an object (in this case using a printable surrogate identifier) and to "navigate" through the schema by applying attributes directly to that object. In the relational model, on the other hand, users must navigate through the schema within the provided record structure using joins. In the SEQUEL language, for example, the analogous query directed at the schema of Figure 2 would be

```
select CITY
from BUSINESS
where BNAME in
  select EMPLOYER
  from BUSTRAV
  where PNAME = 'Mary'
```

In essence, the user first obtains the name of Mary's employer by selecting the record about Mary in the relation BUSTRAV and retrieving the EMPLOYER attribute, then finds the record in the relation BUSINESS that has that value in its BNAME field, and finally reads the CITY attribute of that record. Thus, the linkage between the BUSTRAV and BUSINESS relations is obtained by explicitly comparing business identifiers (the EMPLOYER coordinate of BUSTRAV and the BNAME coordinate of BUSINESS).

1.3.2 Semantic Overloading

The second fundamental advantage cited for the semantic models focuses on the fact that the record-oriented models provide only two or three constructs for representing data interrelationships, whereas semantic models typically provide several such constructs. As a result, constructs in record-oriented models are *semantically overloaded* in the sense that several different types of relationships must be represented using the same constructs [Hammer and McLeod 1981; Kent 1978, 1979; Smith and Smith 1977; Su 1983]. In the relational model, for example, there are only two ways of representing relationships between objects: (1) within a relation and (2) by using the same values in two or more relations.

To illustrate this point, we briefly compare the relational and semantic schemas of the World Traveler database. In the relational schema, at least three different types of relationships are represented structurally within individual relations:

(1) the functional relationship between PNAME and STREET;
(2) the many–many association between PNAMEs and LANGUAGEs;
(3) the clustering of STREET, CITY, and ZIP values as addresses.

At least three other types of relationships are represented by pairs of relations:

(a) the type/subtype relationship between PERSON and TOURIST;
(b) the fact that PERSON, PERSPEAKS, and PERGOES all describe the same set of objects;
(c) the fact that the employers of BUSTRAVs are described in the BUSINESS relation.

In contrast, each of these types of relationship has a different representation in the semantic schema.

As indicated above, in the absence of integrity constraints the data structuring

primitives of the relational model (and the other record-oriented models) are not sufficient to model the different types of commonly arising data relationships accurately. This is one reason that integrity constraints such as key and inclusion dependencies are commonly used in conjunction with the relational model. Although these do provide a more accurate representation of the data, they are typically expressed in a text-based language; it is therefore difficult to comprehend their combined significance. A primary objective of many semantic models has been to provide a coherent family of constructs for representing in a structural manner the kinds of information that the relational model can represent only through constraints. Indeed, semantic modeling can be viewed as having shifted a substantial amount of schema information from the constraint side to the structure side.

1.3.3 Abstraction Mechanisms

Semantic models provide a variety of convenient mechanisms for viewing and accessing the schema at different levels of abstraction [Hammer and McLeod 1981; King and McLeod 1985a; Smith and Smith 1977; Su 1983; Tsichritzis and Lochovsky 1982]. One dimension of abstraction provided by these models concerns the level of detail at which portions of a schema can be viewed. On the most abstract level, only object types and ISA relationships are considered. At this level the structure of objects is ignored; for example, the x-node ADDRESS would be shown without its children. A more detailed view includes the structure of complex objects; the further detail includes attributes and the rules governing derived schema components.

A second dimension of the abstraction provided by semantic models is the degree of modularity they provide. It is easy to isolate information about a given type, its subtypes, and its attributes. Furthermore, it is easy to follow semantic connections (e.g., attribute and ISA relationships) to find closely associated object types. Both of the above dimensions of abstraction are very useful in schema design and for schema *browsing*, that is, the ad hoc perusal of a schema to determine what and how things are modeled. Interactive graphics-based systems that use these properties of semantic models have been developed (see Section 4.3); comparable systems for the record-oriented models have not been developed.

An interesting question is why the central components of semantic models—objects, attributes, ISA relationships—are necessarily the best mechanisms to use to enrich a data model. Although, of course, there can be no clearcut choice of modeling constructs, there are two reasons to support the selection of these particular primitives. First, practice has shown that schemas constructed with traditional record-oriented models tend to simulate objects and attributes by interrelating records of different types with logical and physical pointers. The second point is that computer science researchers in AI and programming languages have selected similar constructs to enhance the usability of other software tools. It is thus interesting that researchers with somewhat different goals have found semantic model-like mechanisms useful. This latter point is discussed in more detail later in this section.

A third dimension of abstraction is provided by derived schema components that are supported by a few semantic models [Hammer and McLeod 1981; King and McLeod 1985a; Shipman 1981] and also by some relational implementations [Stonebraker et al. 1976]. These schema components allow users to define new portions of a schema in terms of existing portions of a schema. Derived schema components permit the user to identify a specific subset of the data, possibly perform computations on it, and then structure it in a new format. The "new" data are then given a name and can subsequently be used while ignoring the details of the computation and reformatting. In the relational model, derived schema components must be either new relations or new columns in existing relations. Semantic models provide a much richer framework for defining derived schema components. For example, a derived subtype specifies both a new type and

an ISA relationship; similarly, a derived single-valued attribute specifies both a piece of data and a constraint on it. Therefore, semantic models give the user considerably more power for abstracting data in this way.

Derived data are closely related to the notion of a *user view* (or external schema) [Chamberlain et al. 1975; Tsichritzis and Klug 1977], except that derived data are incorporated directly into the original schema rather than used to form a separate new schema. Another difference is that a view may contain raw or underived components, as well as derived information.

1.4 Database Design with a Semantic Model

In general, the advantages of semantic models, as described in the literature, are oriented toward the support of database design and evolution [Brodie and Ridjanovic 1984; Chen 1976; King and McLeod 1985a; Smith and Smith 1977]. At the present time the practical use of semantic models has been generally limited to the design of record-oriented schemas. Designers often find it easier to express the high-level structure of an application in a semantic model and then map the semantic schema into a lower level model. One prominent semantic model, the Entity–Relationship Model, has been used to design relational and network schemas for over a decade [Teorey et al. 1986]. Interestingly, relational schemas designed using the ER Model are typically in 3NF, an indication of the naturalness of using a semantic model as a design tool for traditional DBMSs.

A number of features of semantic models contribute to their use in both the design and the eventual evolution of database schemas. They provide constructs that closely parallel the kinds of relationships typically arising in database application areas; this makes the design process easier and lessens the likelihood of design errors. This is in contrast to record-oriented models, which force the designer to concentrate on many low-level details. Semantic models also provide a variety of abstraction mechanisms that researchers have used to develop structured design methodologies. A detailed and fairly comprehensive design methodology appears in Rosussopoulos and Yeh [1984]. After requirements analysis is performed, the authors advise the use of a semantic model as a means of integrating and formalizing the requirements. A semantic model serves nicely as a buffer between the form of requirements collected from noncomputer specialists and the low-level computer-oriented form of record-oriented models. Several methodologies have also addressed the issue of integrating schema and transaction design in order to simplify the collection and formalization of database dynamic requirements; see Brodie and Ridjanovic [1984] and King and McLeod [1985a] for examples.

Semantic models are a convenient mechanism for allowing database specifications to evolve incrementally in a natural, controlled fashion [Brodie and Ridjanovic 1984; Chen 1976; King and McLeod 1985a; Teorey 1986]. This is because semantic models provide a framework for top-down schema design, beginning with the specification of the major object types arising in the application environment, then specifying subsidiary object types. Referring to the World Traveler schema, the design might begin with the specification of the PERSON and BUSINESS nodes; the LINGUIST, TOURIST, and BUSINESS-TRAVELER nodes would follow; and finally the various attributes would be defined. The constructed type ADDRESS might be introduced when it is realized that both PERSON and BUSINESS share the identical attributes STREET, CITY, and ZIP.

In conclusion, significant research has been directed at applying specific semantic models to the design of either semantic or traditional database schemas. However, little work has been directed at providing methodological support for selecting an appropriate semantic model or for integrating the various modeling capabilities found in semantic models. Rather, methodological approaches are typically tied to one model and to one prescriptive approach to producing a semantic schema.

1.5 Related Work in Artificial Intelligence

We now consider the relationship between semantic data modeling and research on knowledge representation in artificial intelligence. Although they have different goals, these two areas have developed similar conceptual tools.

Early research on knowledge representation focused on *semantic networks* [Findler 1979; Israel and Brachman 1984; Mylopoulos 1980] and *frames* [Brachman and Schmolze 1985; Fikes and Kehler 1985; Minsky 1984]. In a semantic network, real-world knowledge is represented as a graph formed of data items connected by edges. The graph edges can be used to construct complex items recursively and to place items in categories according to similar properties. The important relationship types of *ISA*, *is-instance-of*, and *is-part-of* (which is closely related to aggregation) are naturally modeled in this context. Unlike semantic data models, semantic networks mix schema and data in the sense that they do not typically provide convenient ways of abstracting the *structure* of data from the data itself. As a consequence, each object modeled in a semantic network is represented using a node of the semantic network; these networks can be quite large if many objects are modeled. One of the earliest semantic database models, the Semantic Binary Data Model [Abrial 1974], is closely related to semantic networks; schemas from this model are essentially semantic networks that focus exclusively on object classes.

Frame-based approaches provide a much more structured representation for object classes and relationships between them. Indeed, there are several rough parallels between the frame-based approach and semantic data models. The frame-based analog of the abstract object types is called a *frame*. A frame generally consists of a list of *properties* of objects in the type (e.g., elephants have four legs) and a tuple of *slots*, which are essentially equivalent to the attributes of semantic data models. Frames are typically organized using ISA relationships, and slots are inherited along ISA paths in a manner similar to the semantic data models. In general, properties of a type are inherited by a subtype, but exceptions to this inheritance can also be expressed within the framework (e.g., three-legged elephants are elephants, but have only three legs). Exception-handling mechanisms may also be provided for the inheritance of slot values. For example, referring to the World Traveler Database, in a frame-based approach the HAS-NAME attribute of a *given person might be different in the role* of PERSON and the role of TOURIST (e.g., a nick-name). (Although the terminology used by the KL-ONE model [Brachman and Schmolze 1985] differs from that just given, essentially the same concepts are incorporated there.)

In general, frame-based approaches do not permit explicit mechanisms, such as aggregation and grouping for object construction. In recent research and commercial systems [Aikens 1985; Kehler and Clemenson 1983; Stefik et al. 1983], frames have been extended so that slots can hold methods in the sense of object-oriented programming languages; this development parallels current research in object-oriented databases, which is briefly discussed in Section 5.

Because frame-based systems are generally in-memory tools, the sorts of research efforts that have been directed at *implementing semantic databases have not been* applied to them. For example, considerable research effort has focused on the efficient implementation of semantic schemas and derived schema components [Chan et al. 1982; Farmer et al. 1985; Hudson and King 1986, 1987; Smith et al. 1981].

2. TUTORIAL

This section provides an in-depth discussion of the fundamental features and components common to most semantic database models. The various building blocks used in semantic models are described and illustrated, and subtle and not-so-subtle differences between similar components are highlighted. Philosophical implications of the overall approaches to modeling taken by different models are also considered.

To provide a basis for our discussion, we use the Generic Semantic Model (GSM). The model was developed expressly for this survey and is based largely on three of the most prominent models found in the literature: the Entity–Relationship (ER) Model, the Functional Data Model (FDM), and the Semantic Data Model (SDM). The GSM is derived in large part from the IFO Model [Abiteboul and Hull 1987], which itself was developed as a theoretical framework for studying the prominent semantic models [Abrial 1974; Brodie and Ridjanovic 1984; Hammer and McLeod 1981; Kerschberg and Pacheco 1976; King and McLeod 1985a; Shipman 1981; Sibley and Kerschberg 1977]. Although the GSM incorporates many of the constructs and features of these models, it cannot be a true integration of all semantic models because of the very different approaches they take. Specifically, the approach taken by GSM is closest to the FDM. Because the primary purpose of GSM has been to serve as a tool for exposition, it is not completely specified in this paper.

In some cases the literature taken as a whole uses a given term ambiguously. Perhaps the most common example of this is the term "aggregation." At a philosophical level, this term is used universally to indicate object types that are formed by combining a group of other objects; for example, ADDRESS might be modeled as an aggregation of STREET, CITY, and ZIP. At a more technical level, some models support this using a construction based on Cartesian product, whereas others use a construction based on attributes. In this section we adopt specific, somewhat technical definitions for various terms. For example, we use aggregation to refer to Cartesian-product-based constructions. These more restrictive definitions will permit a clear articulation of the different concepts arising in the literature.

This section has four major parts. The first briefly compares two broad philosophical approaches that many models choose between, providing a useful perspective before delving into a detailed discussion of the different building blocks of semantic models. The second part defines the specific constructs used for describing the *structure* of data in semantic models and presents examples that highlight similarities and differences between them. The third considers how these constructs are combined and augmented to form database schemas in semantic models. The fourth discusses languages for accessing and manipulating data, and for specifying semantic schemas.

2.1 Two Philosophical Approaches

The GSM is meant to be representative of a wide class of semantic models; as a result of being somewhat eclectic, it blurs an important philosophical distinction arising in semantic modeling literature. Historically, there have been two general approaches taken in constructing semantic models. The distinction between them is not black and white, but models have had a tendency to adopt one approach or the other. Essentially, various models place different emphasis on the various constructs for interrelating object classes. One approach stresses the use of attributes to interrelate objects; the other places an emphasis on explicit type constructors. As a result, different data models may yield dramatically different schemas for the same underlying application.

To illustrate this point, for the same underlying data we compare two schemas that give very different prominence to attributes and type constructors. The comparison is particularly salient because the schemas reflect the underlying philosophies of two early influential semantic models, namely, the FDM and the ER Models, respectively.

Figure 3 shows the two GSM schemas, both representing the same data underlying a portion of the World Traveler Database application. The schema in Figure 3a loosely follows the FDM and emphasizes the use of attributes for relating abstract object types with other abstract object types. The schema in Figure 3b loosely follows the philosophy of the ER Model in that it emphasizes the use of type constructor aggregation (called *relationship* in the ER Model) and grouping for relating

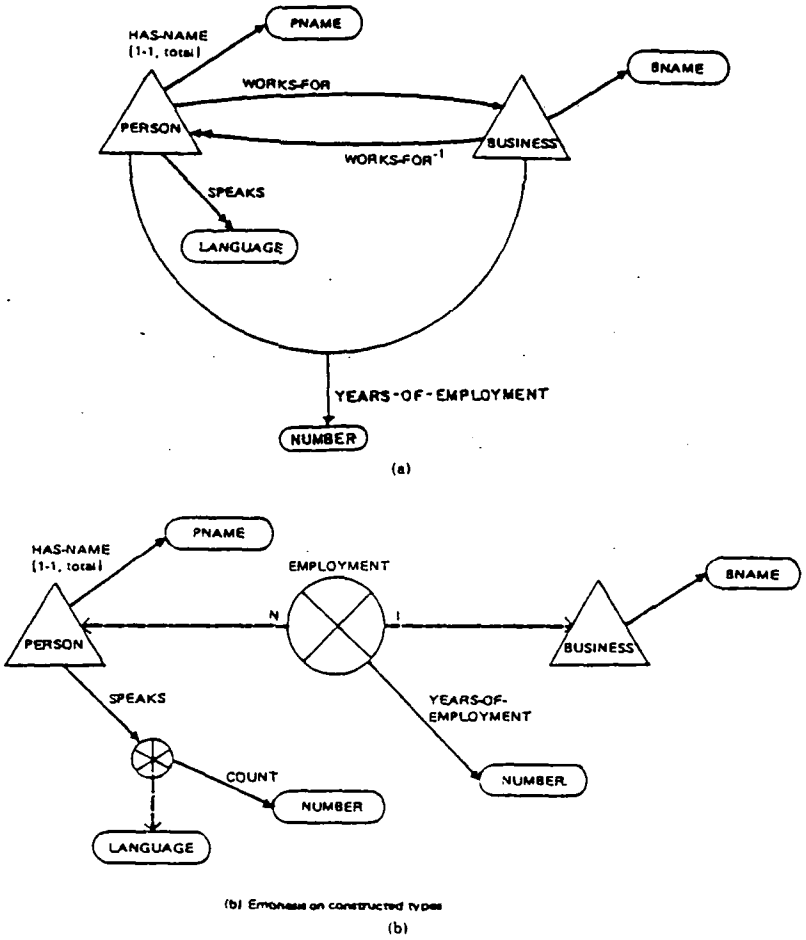


Figure 3. Two schemas for the same underlying data. (a) Schema emphasizing attributes. (b) Schema emphasizing type constructors.

abstract object types. In both schemas an instance includes a set of PERSONs and a set of BUSINESSes (both considered sets of abstract objects), along with attributes specifying person and business names and the languages spoken by PERSONs.

Interestingly, in an instance of the first schema the relationship of people and their business is represented by the attribute (i.e., function) WORKS-FOR and its inverse WORKS-FOR^{-1}; in the second, the aggregation EMPLOYMENT (which is a

set of ordered pairs) is used. Both schemas represent the constraint that many people work for the same business, but not the reverse: In the first schema this is accomplished using a single-valued and a multivalued attribute, and in the second by the N:1 constraint. Further, in the first schema, a multivalued attribute is used to represent the languages spoken by a person, whereas in the second, a grouping construct is used.

The choice of emphasis—attribute based or type constructor based—affects the language mechanisms that seem natural for manipulating semantic databases. Consider Figure 3a. If a user wanted to know the business of a particular person, the attribute WORKS-FOR may be used to reference the business directly. In Figure 3b, the type constructor representing ordered pairs of PERSONs and BUSINESSes must be manipulated in order to obtain the desired data. On the other hand, the type constructor approach gives the user the flexibility of directly referencing, by name, ordered pairs in EMPLOYMENT.

The use of type constructors also allows information to be associated directly with schema abstractions. As one illustration, the bottom subschema includes an attribute on EMPLOYMENT that describes the length of time an individual has been employed at a particular company. (Essentially the same information is represented in the first schema with the two-argument attribute YEARS-OF-EMPLOYMENT, although the relationship EMPLOYMENT and this attribute are not linked together.) Analogously, in the second schema, the grouping construct for LANGUAGES is augmented by an attribute giving the cardinality of each set of languages. (No analog for this exists in the attribute-based approach.) In a model that stresses type constructors, relationships between types are essentially viewed as types in their own right; thus it makes perfect sense to allow these types to have attributes that further describe them.

2.2 Local Constructs

This section presents detailed descriptions of the building blocks that semantic models use to represent the structure of data. The discussion is broken into three parts, which focus on types, attributes, and ISA relationships, respectively. Importantly, in the section on attributes we compare the notions of attributes and aggregations.

2.2.1 *Atomic and Constructed Types*

A fundamental aspect of all semantic models is the direct representation of object types, distinct from their attributes and sub- or supertypes. Most models provide mechanisms to represent atomic or nonconstructed object types, and many models also provide type constructors. In the discussion below we focus on the use of object types in semantic models and on the two most prominent type constructors, namely, aggregation and grouping.

A semantic model typically provides the ability to specify a number of *atomic types*. Intuitively, each of these types corresponds to a class of nonaggregate objects in the world, such as PERSONs or ZIP-codes. (Of course, the type *PERSON has many attributes*.) Many semantic models distinguish between atomic types that are *abstract* and those that are *printable* (or *representable*). The abstract types are typically used for physical objects in the world, such as PERSONs, and for conceptual (or legal) objects, such as BUSINESSes. Atomic printable types are typically alphanumeric strings, but in some graphics-based systems they might include icons as well. It is often convenient to articulate subclasses of these, such as ZIP-codes, Person-NAMEs, or Business-NAMEs, and most models associate operators, such as addition for numbers, with them. As shown in the World Traveler schema, in the GSM abstract types are depicted with triangles, atomic printable types are depicted with flattened ovals, and subtypes are depicted with circles.

In instances of a semantic schema, abstract objects are viewed conceptually to correspond directly to physical or conceptual objects in the world and in some implementations of semantic models, they are represented using internal identifiers that are not directly accessible to the user. This corresponds to the intuition that such

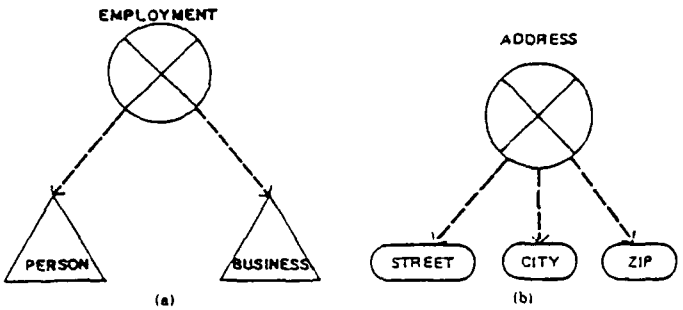


Figure 4. Object types constructed with aggregation. (a) EMPLOYMENT = PERSON × BUSINESS. (b) ADDRESS = STREET × CITY × ZIP.

objects cannot be "printed" or "displayed" on paper or on a monitor.

When defining an instance of a semantic schema, an *active domain* is associated with each node of the schema. The active domain of an atomic type holds all objects of that type that are currently in the database. This notion of active domain is extended to type constructor nodes below.

We now turn to *type constructors*. The most prominent of these in the semantic literature are *aggregation* (called *relationship* in the ER Model) and *grouping* (also known as *association* [Brodie and Ridjanovic 1984]). An aggregation is a composite object constructed from other objects in the database. For example, each object associated with the aggregation type EMPLOYMENT in Figure 4a is an ordered pair of PERSON and BUSINESS values. Mathematically, an aggregation is an ordered n-tuple. In an instance, the active domain of an aggregation type will be a *subset* of the Cartesian product of the active domains assigned to the underlying nodes. For example, the active domain of EMPLOYMENT will be the set of pairs corresponding to the set of employee–employer relationships currently true in the database application. According to our definition, the identity of an aggregation object is completely determined by its component values. Figure 4b highlights the use of aggregation for encapsulating information.

Before continuing, we reiterate that the definition of aggregation used here is deliberately narrow and differs from the usage of that term in some models, including SDM and TAXIS. The representation of aggregations in those models is generally based on attributes and is discussed in the next section. It should also be noted that some models, including FDM, emphasize the use of attributes, as well as support the use of aggregations in attribute domains.

The grouping construct is used to represent sets of objects of the same type. Figure 5a shows the GSM depiction of the grouping construct to form a type whose objects are sets of languages. Mathematically, a grouping is a finite set. In an instance, the active domain of a grouping type will hold a set of objects, each of which is a finite subset of the active domain of the underlying node. In a constructed object, a ∩-node will always have exactly one child.

As defined here, a grouping object is a set of objects. Technically, then, the *identity of a grouping object is determined* completely by that set. To emphasize the significance of this, we consider how committees might be modeled in a semantic schema. One approach is to define the type COMMITTEE as a grouping of PERSON because each committee is basically a set of people. This is probably not accurate in most cases because the identity of a

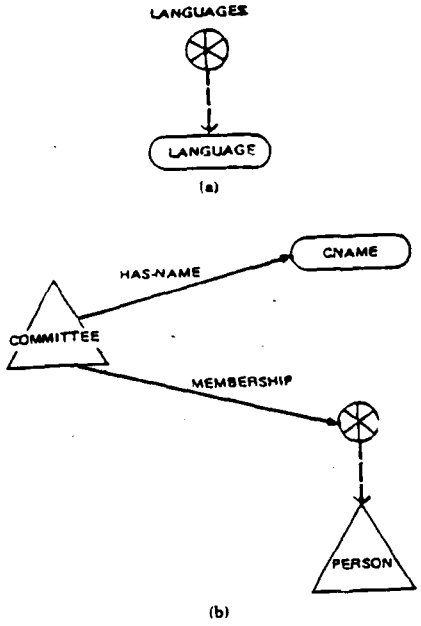


Figure 5. Object types constructed with grouping. (a) LANGUAGES = • LANGUAGE.

committee is separate from its membership at a particular time. Figure 5b shows a more appropriate approach. COMMITTEE is modeled as an abstract type and has an attribute MEMBERSHIP whose range is a grouping type.

As illustrated in Figure 6, the type constructors can be applied recursively. In this example, we view a VISIT as a triple consisting of a TOURIST-TRAP, a GUIDE (viewed as a subtype of PERSON), and a set of TOURISTs (also a subtype of person). As indicated in the figure, edges originating from an aggregation node can be labeled by a role; this is important if more than one child of an aggregation is of the same type. In the GSM and most semantic models supporting aggregation and grouping, there can be no (directed or undirected) cycle of type constructor edges. The Logical Data Model [Kuper and Vardi 1984, 1985] provides an alternative formalism in which cycles are permitted.

We close this section by mentioning other kinds of type constructors found in the literature. The TAXIS and Galileo models support *metatypes*; that is, types whose elements are themselves types. For example, in the World Travele xample, a metatype TYPE-OF-PERSON might contain the types PERSON, LINGUIST, TOURIST, and BUSINESS-TRAVELER. This metatype could have attributes such as SIZE or AVERAGE-AGE, which describe characteristics of the populations of the underlying types. A comparison of metatypes with both subtypes and the grouping construct is presented in Section 2.3.2.

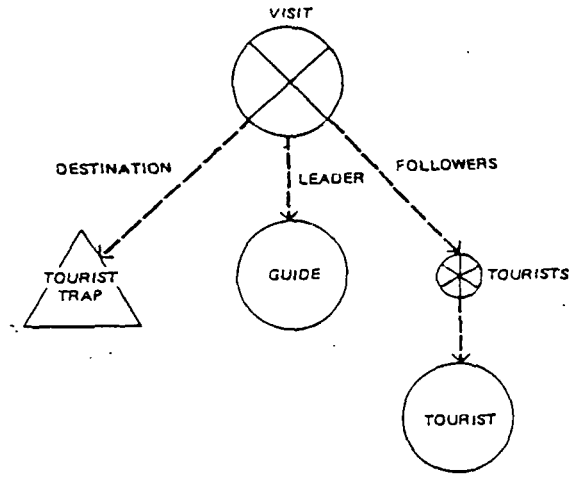
In principle, a data model can support essentially any type constructor in much the same way in which some programming languages do. Historically, almost all semantic models have focused almost exclusively on aggregation and grouping. Notable exceptions include SAM* (Semantic Association Model), TAXIS, and Galileo. These models permit a variety of type constructors that may be applied to atomic printable types. SAM* is oriented in part toward scientific and statistical applications and supports sets, vectors, ordered sets, and matrices; TAXIS and Galileo supports type constructors typical of imperative programming languages.

To summarize, semantic models typically differentiate between abstract and printable types and provide type constructors for aggregation and grouping.

2.2.2 *Attributes*

The second fundamental mechanism found in semantic models for relating objects is the notion of attribute (or function) between types. In this section we articulate a specific meaning for this notion and indicate the various forms it takes in different semantic models. We conclude with a comparison of different modeling strategies using aggregation and attributes.

We begin by defining the notion of attribute as used in the GSM. Speaking formally,



VISIT = DESTINATION:TOURIST-TRAP × LEADER:GUIDE × FOLLOWERS:(•TOURIST)

Figure 6. Recursive application of aggregation and grouping constructs.

a *one-argument attribute* in a GSM schema is a directed binary relationship between two types (depicted by an arrow), and an *n-argument attribute* is a directed relationship between a set of n types and one type (depicted by an arrow with n tails). Attributes can be *single valued*, depicted using an arrow with one pointer at its head, or *multivalued*, depicted using an arrow with two pointers at its head. In an instance, a mapping (a binary or $(n + 1)$-ary relation) is assigned to each attribute; the domain of this mapping is the (cross product of the) active domain(s) of the source(s) of the attribute, and the range is the active domain of the target of the attribute. The mapping may be specified explicitly through updates, or in the case of derived attributes it may be computed according to a derivation rule. In the case of a single-valued attribute, the mapping must be a function in the strict mathematical sense, that is, each object (or tuple) in the domain is assigned at most one object in the range. In GSM, there are no restrictions on the types of the source or target of an attribute.

Of course, there is a close correspondence between the semantics of a multivalued attribute and the semantics of a single-valued attribute whose range is a constructed groupin r type. In keeping with the general philosophy that the GSM incorporates prominent features from several representative semantic models, both of these possibilities have been included. Most models in the literature support multivalued attributes and do not permit an attribute to map to a grouping type. Also, some models, including SDM and INSYDE, view all attributes as multivalued and use a constraint if one of them is to be single valued. Similarly, there is also a close relationship between a one-argument attribute whose domain is an aggregation and an n-argument attribute.

We now briefly mention another kind of attribute, called here a *type attribute*. This is supported in several models, including SDM, TAXIS, and SAM*. Type attributes associate a value with an entire type, instead of associating a value with each object in the active domain. For example,

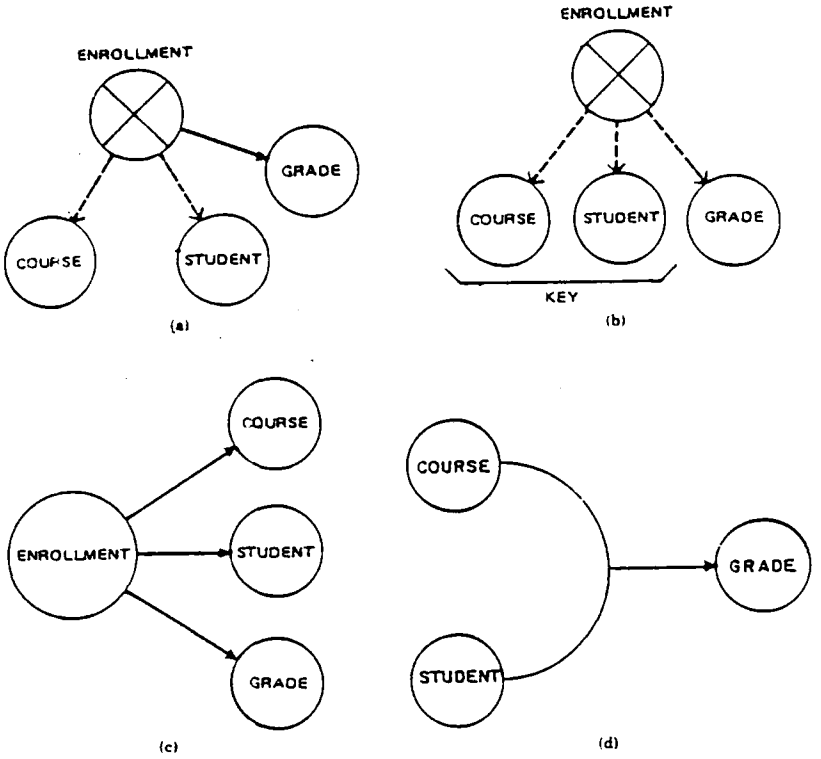


Figure 7. Four alternative representations for ENROLLMENT.

the type attribute COUNT might be associated with the type PERSON and would hold one value: the number of people currently "in" the database. Other type attributes might hold more complex statistics about a type, for example, the average salary or the standard deviation of those salaries. The value associated with a type attribute is generally prescribed in the schema; such attributes thus form a special kind of derived data.

We conclude the section by comparing four different ways of representing essentially the same data interrelationships using the aggregation and attribute constructs. Figure 7 shows four subschemas that might be used to model the type ENROLLMENT. To simplify the pictures, we depict all atomic nodes as circular. In the first subschema, ENROLLMENT is viewed as an aggregation of COURSE and STUDENT. Each object of type ENROLLMENT will be an ordered pair, and a GRADE is associated with it by the attribute shown. The IFO and Galileo models provide explicit mechanisms for this representation. The second approach might be taken in such models as SAM* and SHM+, which do not provide an explicit attribute construct. In this case ENROLLMENT is

viewed as a ternary aggregation of COURSE, STUDENT, and GRADE. As suggested in the diagram, a key constraint is typically incorporated into this schema to ensure that each course–student pair has only one associated grade. The third approach shown in Figure 7c might be taken in models that do not provide an explicit type constructor for aggregation. Many semantic models fall into this category, including SBDM, SDM, TAXIS, and INSYDE (and the object-oriented programming language SMALLTALK, for that matter). Under this approach ENROLLMENT is viewed as an atomic type with three attributes defined on it. Although not shown in Figure 7c, a constraint might be included so that no course–student pair has more than one grade. The fourth approach is especially interesting in that it does not require that the construct ENROLLMENT be explicitly named or defined if it is not in itself relevant to the application. In this case the attribute for GRADE would be a function with two arguments. FDM has this capability.

We now compare the first three of these approaches from the perspective of object identity. In Figure 7a, each enrollment is an ordered pair. Thus, the grade associated with an enrollment can change without affecting the identity of the enrollment. Technically speaking, in the absence of the key dependency, this is not true in Figure 7b, in which an enrollment is an ordered triple. In Figure 7c, the underlying identity is independent of any of the associated course, student, and grade values. An enrollment e with values CS101, Mary, and 'A' might be modified to have values Math2, Mary, 'B' without losing its underlying identity. Also, in the absence of a constraint, the structure does not preclude the possibility that two distinct enrollments e and e' have the same course, the same student, and the same grade.

2.2.3 ISA Relationships

The third fundamental component of virtually all semantic models is the ability to represent *ISA* or supertype/subtype relationships. In this section we review the basic intuitions underlying these relationships and describe different variations on the concept found in the literature. The focus of this section is on the local properties of ISA relationships; global restrictions on how they may be combined are discussed in Section 2.3.1. In several models subtypes arise almost exclusively as derived subtypes; this aspect of subtypes is considered in Section 2.3.2.

Intuitively, an ISA relationship from a type SUB to a type SUPER indicates that each object associated with SUB is associated with the type SUPER. For example, in the World Traveler schema the ISA edge from TOURIST to PERSON indicates that each tourist is a person. More formally, in each instance of the schema, the active domain of TOURIST must be contained in the active domain of PERSON. In most semantic models each attribute defined on the type SUPER is automatically defined on SUB; that is, attributes of SUPER are *inherited* by SUB. It is also generally true that a subtype may have attributes not shared by the parent type.

The family of ISA relationships in a schema forms a directed graph. In the literature this has been widely termed the ISA "hierarchy." However, as suggested in Figure 8, most semantic models permit undirected (or weak) cycles in this graph. For this reason we follow Atzeni and Parker [1986] and Lenzerini [1987] in adopting the term *ISA network*. Although ISA relationships are transitive, it is customary to specify the fundamental ISA relationships explicitly and view the links due to transitivity as specified implicitly.

Speaking informally, ISA relationships might be used in a semantic schema for two closely related purposes. The first is to represent one or more possibly overlapping subtypes of a type, as with the subtypes of PERSON shown in the World Traveler schema. The second purpose is to form a type that contains the union of types already present in a schema. For example, a type VEHICLE might be defined as the union of the types CAR, BOAT, and PLANE, or the type LEGAL-ENTITY might be the union of PERSON, CORPORATION, and LIMITED-PARTNER

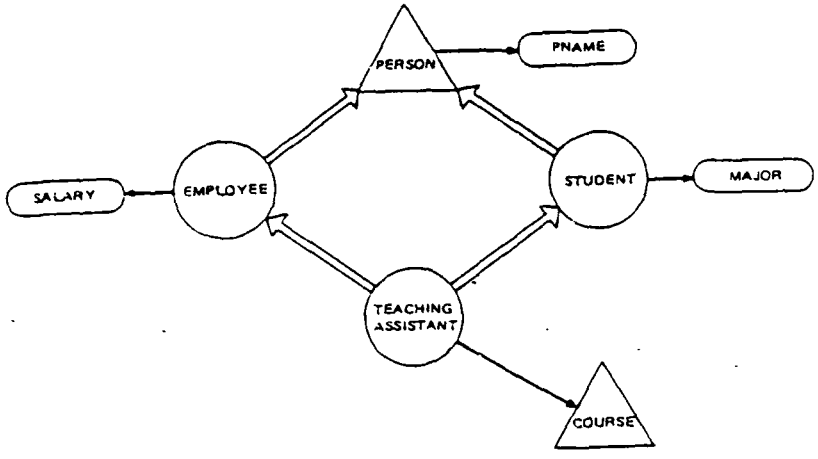


Figure 8. ISA network with undirected cycle.

SHIP. When using ISA for forming a union, it is common to include a *covering constraint*, which states that the (active domain of the) supertype is contained in the union of the (active domains of the) subtypes. Also, the semantics of update propagation varies for the different kinds of ISA relationships.

Historically, semantic models have used a single kind of ISA relationship for both of these purposes. Furthermore, several early papers on semantic modeling (including FDM and SDM) provide schema definition primitives that favor the specification of ISA networks from top to bottom. For example, in these models the type VEHICLE would be specified first, and subtypes CAR, BOAT, and PLANE would be specified subsequently. In contrast, the seminal paper [Smith and Smith 1977] uses ISA relationships to form unions of existing types.

More recent research on semantic modeling has differentiated several kinds of ISA relationship: and some models, including IFO, RM/T, Galileo, and extensions of the ER Model, incorporate more than one type of ISA into the same model. For example, in the extension of the ER Model described in Teorey et al. [1986], *subset* and *generalization* ISA relationships are supported. A subset ISA relationship arises when one type is contained in another: this is the notion already discussed in connection with the GSM. Generalization ISA relationships arise when one type is *partitioned* by its subtypes, that is, when the subtypes are disjoint and together cover the supertype. Generalization ISA relationships could thus be used for the VEHICLE and LEGAL-ENTITY types mentioned above. As noted in Abiteboul and Hull [1987] and Teorey et al. [1986], the update semantics of these two constructs are different. For example, in the first case deletion of an object from a subtype has no impact on the supertype; in the second case deletion from a subtype also requires deletion from the supertype.

A second broad motivation for distinguishing kinds of ISA relationships stems from studies of schema integration [Batini et al. 1986; Dayal and Hwang 1984; Navathe et al. 1986; NEL86]. For example, Dayal and Hwang [1984] study the problem of integrating two or more FDM schemas. Suppose that two FDM schemas contain types EMP1 and EMP2, respectively, for

employees. To integrate these, a new type EMPLOYEE can be formed as the generalization of EMP1 and EMP2. This generalization may have overlapping subtypes but must be covered by them. Interestingly, Dayal and Hwang [1984] also permit ISA relationships between attributes.

2.3 Global Considerations

In Section 2.2 we discussed the constructs used in semantic models largely in isolation. This section takes a broader perspective and examines the larger issue of how the constructs are used to form schemas. The discussion is broken into three areas. The first concerns restrictions of an essentially structural nature on how the constructs can be combined, for example, that there be no directed cycles of ISA relationships. The second and third areas are two closely related mechanisms for extending the expressive power of schemas, namely, derived schema components and integrity constraints.

2.3.1 Combining the Local Constructs

Although many semantic models support the basic constructs of object construction, attribute, and ISA, they do not permit arbitrary combinations of them in the formation of schemas. Restrictions on how the constructs can be combined generally stem from underlying philosophical principles or from intuitive considerations concerning the use or meaning of different possible combinations. Such restrictions have also played a prominent role in theoretical investigations of update propagation in semantic schemas [Abiteboul and Hull 1987; Hecht and Kerschberg 1981]. The restrictions are typically realized in one of two ways: in the definition of the constructs themselves (e.g., in the original ER Model, all attribute ranges are printable types) or as global restrictions on schema formation (e.g., that there be no directed cycles of ISA relationships). The following discussion surveys some of the intuitions and restrictions arising in construct definitions and then considers global restrictions on schema formation.

In the description of the local constr given in Section 2.2, relatively few rest tions are placed on their combination. example, aggregation and grouping can used recursively, and attributes can h arbitrary domain and range types. Inde part of the design philosophy of the G was to present the underlying construct as unrestricted a form as feasible in or to separate fundamental aspects of the c structs from their usage in the vari semantic models of the literature. In c trast with the GSM, many semantic mod in the literature present constructs restricted forms; for example, some mod permit aggregations in attribute doma but not as attribute ranges or in ISA re tionships.

Restrictions explicitly included in definition of constructs are essentia local. However, these restrictions can aris the overall or global structure of the fam of schemas of a given model. A drama illustration of this is provided by the or nal ER Model [Chen 1976]. In that mod aggregation can be used only to combi abstract types. As a result, schemas in the model have a two-tier character: w abstract types in one level and aggregatio in the second. Attributes may be defin on both abstract types or aggregations. they must have ranges of printable type.

We conclude our discussion of local co structs by attempting to indicate why c tain models introduce restrained versio of constructs. Intuitively, a model design tries to construct a simple yet comprehe sive model that can represent a large fam of naturally occurring applications. Th for example, FDM allows grouping only attribute ranges. As illustrated in the di cussion of COMMITTEEs in Section 2. (see Figure 5b), grouping objects are rar of interest in isolation.

In addition to restricting the use of co structs at the local level, many seman models specify global restrictions on ho they may be combined (including notab Abiteboul and Hull [1987]; Brodie an Ridjanovic [1984]; Brown and Park [1983]; Dayal and Hwang [1984]; Hec and Kerschberg [1981]). The most prom nent restrictions of this kind concern t

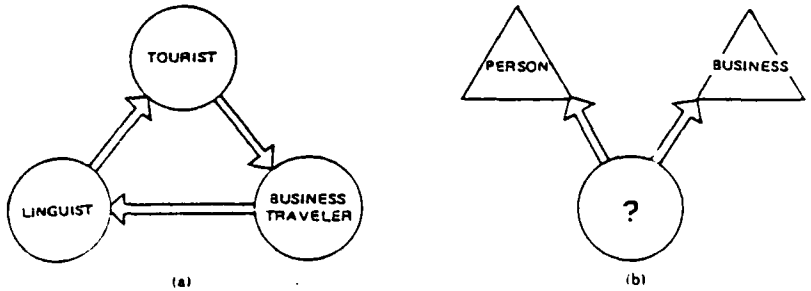


Figure 9. "Schemas" violating intuitions concerning ISA.

combining of ISA relationships. More recently, the interplay between constructed types and ISA relationships has also been studied. To give the flavor of this aspect of semantic models, we present a representative family of global restrictions on ISA relationships. It should also be noted that several models [Albano et al. 1985; Hammer and McLeod 1981; King and McLeod 1985a; Shipman 1981; Su 1983] do not explicitly state global rules of this sort but nevertheless imply them in the definitions of the underlying constructs.

To focus our discussion of ISA restrictions, we consider only abstract types. This coincides with most early semantic models, including FDM and SDM. In schemas for these models, a family of *base types* is viewed as being defined first, and subtypes are subsequently defined from these in a top-to-bottom fashion. The World Traveler schema follows this philosophy, as does the example in Figure 8. In the GSM, subtypes are depicted using a subtype (circle) node, indicating that they are not base types. To enforce this philosophy, we might insist that the tail of each specialization edge is a subtype node and the head of each specialization edge is an abstract or subtype node.

A second general restriction on ISA involves directed cycles. Consider the "schema" of Figure 9a. (We use quotes because this graph does not satisfy the global restriction we are about to state.) It suggests that TOURIST is a subtype of BUSINESS-TRAVELER, which is a subtype of LINGUIST, which is a subtype of TOURIST. Intuitively, this cycle implies that the three types are *redundant*; that is, in every instance, the three types will contain the same set of objects. Furthermore, if the cycle is not connected via ISA relationships to some abstract type, there is no way of determining the underlying type (e.g., PERSON) of any of the three types. Thus, we might insist that there is no directed cycle of ISA edges.

In the "schema" of Figure 9b, the type labeled ? is supposed to be a subtype of the abstract type PERSON and also of the abstract type BUSINESS. If we suppose that the underlying domains of PERSON and BUSINESS are disjoint, then in every instance the node labeled ? will be assigned the empty set. Speaking intuitively, the ? node cannot hold useful information. So, we might insist that any pair of directed paths of ISA edges originating at a given node can be extended to a common node.

The above discussion provides a *complete* family of restrictions on ISA relationships for the GSM considered without type constructors. Speaking informally, the rules are complete because they capture all of the basic natural intuitions concerning how ISA relationships (of the top-to-bottom variety) must be restricted in order to be meaningful. On a more formal level, it can be shown that, if a schema satisfies these rules, then every node will have an unambiguous underlying type, no pair of nodes will be redundant, and every node will be satisfiable in the sense that some instance will assign a nonempty active domain to that node.

The set of rules given above applies to the special case of abstract types and top-to-bottom ISA relationships. As discussed in Section 2.2.3, some models support different kinds of ISA relationships. Furthermore, in some models constructed types can participate in ISA relationships. Specification of global rules in these cases is more involved; the IFO model presents one such set of rules [Abiteboul and Hull 1987].

2.3.2 *Derived Schema Components*

Derived schema components are one of the fundamental mechanisms in semantic models for data abstraction and encapsulation. A derived schema component consists of two elements: a structural specification for holding the derived information and a mechanism for specifying how that structure is to be filled, called a *derivation rule*. (Keeping with common terminology, we refer to derived schema components simply as "derived data.") Derived data thus allow computed information to be incorporated into a database schema.

In published semantic models the most commonly arising kinds of derived data are *derived subtypes* and *derived attributes*. Each of these is illustrated in the World Traveler schema: LINGUIST is a derived subtype of PERSON that contains all persons who speak at least two languages, and LANG-COUNT is a derived attribute that gives the number of languages that members of LINGUIST speak. In queries, users may freely access these derived data in the same manner in which they access data from other parts of the schema. As a result, the specific computations used to determine the members of LINGUIST and the value of LANG-COUNT are invisible to the user. The derivation rules defining derived data can be quite complex, and moreover, they can use previously defined derived data.

In any given semantic model, a language for specifying derivation rules must be defined. In the notable models supporting derived data [Hammer and McLeod 1981; King and McLeod 1985a; Shipman 1981], this language is a variant of the first-order predicate calculus, extended to permit the direct use of attribute names occurring in the schema, the use of aggregate attributes, and the use of set operators (such as set membership and set inclusion). This is discussed further in Section 2.4. (Although not traditionally done, the language for specifying derivation rules can, in principle, allow side effects.)

To illustrate the potential power of a derived data mechanism, we present an example that could be supported in the DBMS CACTIS [Hudson and King 1986]. Figure 10 shows a schema involving BUSINESS-TRAVELERs and TRIPs they have taken. The derived attribute TOTAL-MILES-TRAVELED is also defined on business travelers. The attribute uses two pieces of information: the TRIP attribute of BUSINESS-TRAVELER and the ADDRESS attribute of BUSINESS. TRIP consists of ordered pairs of DATE and CITY, each representing one business trip. The definition of TOTAL-MILES-TRAVELED is based on a derivation rule that is a relatively complex function. For each city traveled to on a trip, this function computes the distance between that city and the city the individual works in. Then, the distances are summed and multiplied by 2 to give the total miles traveled per individual. This distance information may be stored elsewhere in the database or elsewhere in the system.

To illustrate further the power of derived data, we present an example showing the interplay of derived data with schema structures. The example also provides a useful comparison of the notions of grouping, subtype, and metatype. Figure 11 shows three related ways of modeling categorizations of people on the basis of the languages they can speak. Figure 11a is taken from SDM and uses the grouping construct in conjunction with a derivation rule stating that the node should include sets of people grouped by the languages they speak. In an instance, this type would include the set of persons who speak French, the set of persons who speak

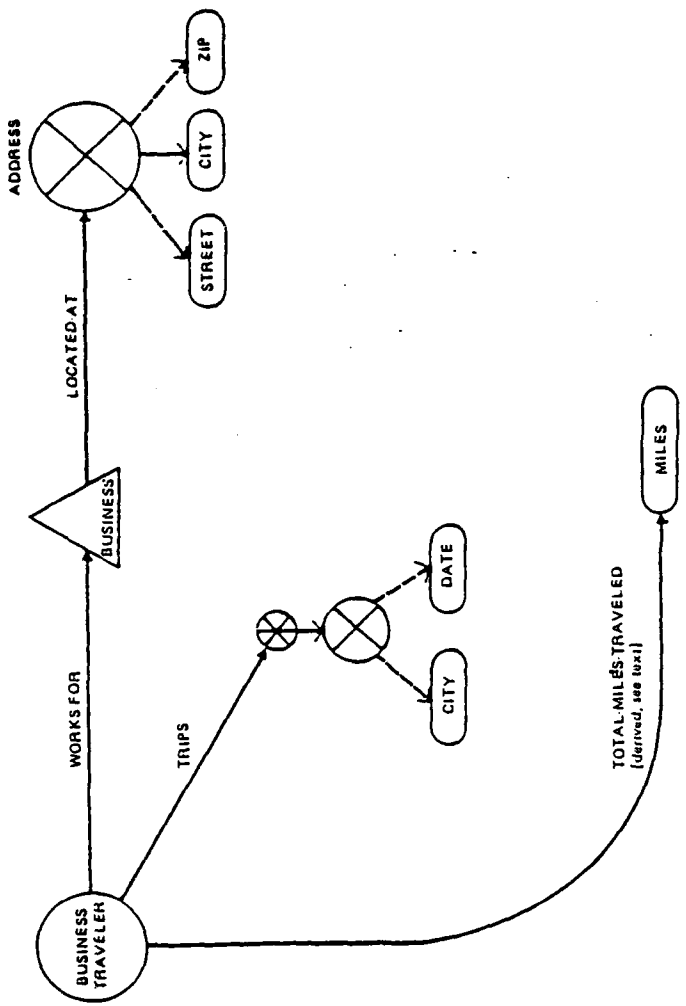


Figure 10. Schema used in example of derived attribute.

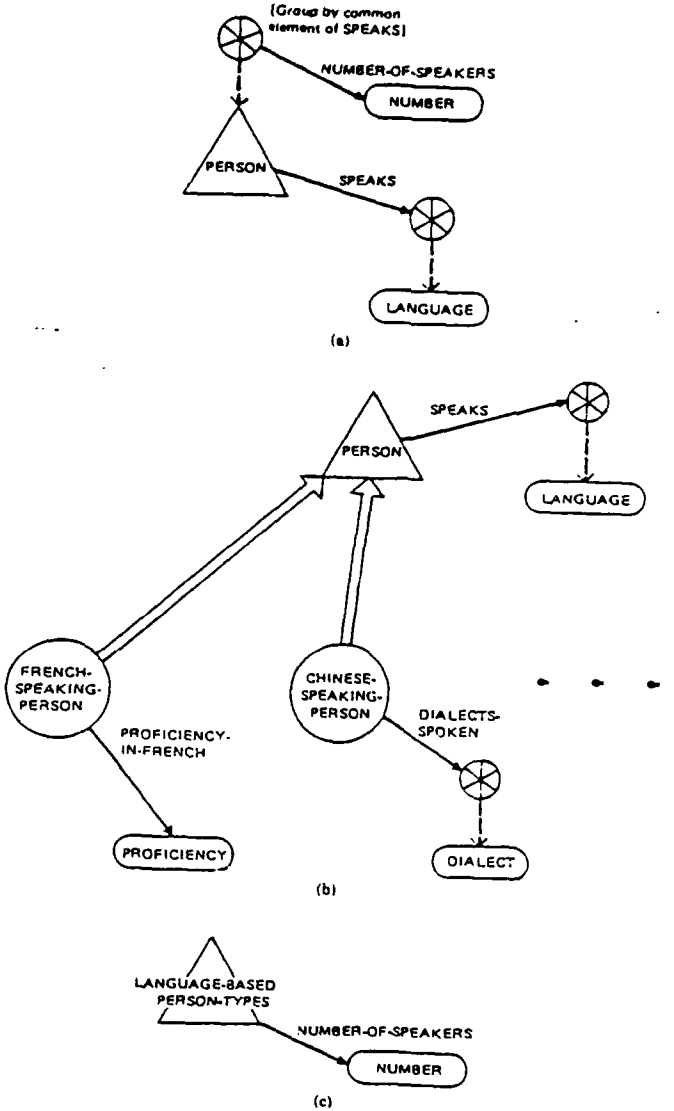


Figure 11. Related uses of derived schema components. (a) Expression-defined grouping type as in SDM. (b) Derived subtypes (derivation rules not shown). (c) Metatype whose elements are types, as in TAXIS.

Chinese, and, more generally, a set of persons for each of the languages in the database. These sets are accessed in queries by referring to languages. (This construction is closely related to forming the inverse function SPEAKS^{-1}.) In the example, we also define a (nonderived) attribute on the grouping type.

The schema of Figure 11b includes a derived subtype for each of the languages that arises. In this representation different attributes can be associated with each of the subtypes. Importantly, the number of subtypes is equal to the number of languages arising in the underlying instance, whereas in the schema of Figure 11a, only one additional type is used. Although not shown here, type attributes can be defined on the subtypes to record information on the number of speakers of each language.

The schema of Figure 11b can be extended to include the graph of Figure 11c, which shows the use of a *metatype*, as found in TAXIS. The elements of this metatype are types from elsewhere in the schema. The derived attribute NUMBER-OF-SPEAKERS defined on this metatype shows a third way of obtaining this cardinality information.

Several models, including FDM and SDM, view the specification of derived data as part of the schema design and/or evolution process, whereas others support a much more dynamic view. For example, in the implementation of INSYDE described in King [1984], users can specify derived data at any time and incorporate them as permanent in the schema. Indeed, in the graphics-based interface to this model [King 1984], database queries are formed through the iterative specification of derived data (see Section 4.3).

We close this section with a discussion of the interaction of derived data with database updates. Speaking in general terms, derived data are automatically updated as required by updates to other parts of the schema. For example, in the World Traveler Database, if a person who speaks one language learns a second, that person is automatically placed in the LINGUIST subtype, and the attribute LANG-COUNT is extended to this person. A subtlety arises if the user attempts to directly update data associated with a derived schema component. In many cases such updates would have ambiguous consequences. For example, in an instance of the World Traveler Database, if someone were explicitly deleted from LINGUIST, the set of languages that person speaks would have to be reduced, but the system would not know which languages to remove.

In some cases explicit updates against a derived schema component might have an unambiguous impact on the underlying data. For example, updates on the FRENCH-SPEAKING-PERSON subtype of Figure 11b are easily translated into updates on the SPEAKS attribute. Importantly, FDM as described in Shipman [1981] provides facilities for specifying how updates to the derived data, if permitted at all, should be propagated in the underlying data. Interestingly, the derived update problem is related to the view update problem in relational databases [Cosmadakis and Papadimitriou 1984].

2.3.3 Static Integrity Constraints

As is clear from the above discussion, the structural component of semantic models provides considerably more expressive power than that of the record-oriented models. However, there is still a wide variety of relationships and properties of relationships that cannot be directly represented using structure alone. For this reason, semantic models often provide mechanisms for specifying integrity constraints. The discussion here focuses on three topics: the relationship between semantic models and the prominent relational integrity constraints, prominent types of integrity constraints found in semantic models, and the differences between integrity constraints and derived data. Although integrity constraints can in principle focus on both the static and dynamic aspects of data [Tsichritzis and Lochovsky 1982; Vianu 1987], little research on dynamic constraints has been done relative to semantic models. For this reason, we focus on static integrity constraints.

Broadly speaking, semantic models express in a structural manner the most

important types of relational integrity constraints, namely, key dependencies and inclusion dependencies. As suggested by the World Traveler schema in Figure 1 and the associated relational schema in Figure 2, relational key dependencies can be represented using single-valued attributes. Inclusion dependencies arising from subtyping can be represented using ISA relationships. Inclusion dependencies that serve as *referential constraints* are typically modeled in an implicit manner in semantic schemas. For example, the dependency BUSTRAV[EMPLOYER] $\subseteq$ BUSINESS[BNAME] in the relational schema is represented in the semantic schema by the fact that the attribute edge WORKS-FOR points directly to the BUSINESS node as its range. Interestingly, some examples of multivalued dependency [Fagin 1977; Zaniolo 1976] are naturally modeled using multivalued attributes.

We now turn to the various kinds of constraints used in semantic models. Many of these focus on restricting the individual constructs occurring in a schema. On attributes, such constraints include restrictions that they be 1–1, onto, or total. For example, in the World Traveler schema, the HAS-NAME attribute is restricted to be 1–1 and total. ISA relationships can also be constrained in various ways. For example, a *disjointness constraint* states that certain subtypes of a type are disjoint (e.g., that no TOURIST is a BUSINESS-TRAVELER). A *covering constraint* states that a set of subtypes together covers a type. In some investigations, these constraints are applied to types that need not be related by ISA edges [Lenzerini 1987].

An important class of constraints on constructs restrict *cardinalities* in various ways. Perhaps the best known types of cardinality constraint are found in the ER Model: These specify whether a binary aggregation (relationship) is 1:1, 1:*N*, *N*:1, or *M*:*N*. For example, in Figure 3b, the aggregation EMPLOYMENT between PERSON and BUSINESS is constrained to be *N*:1. In each instance of this schema, several (*N*) people can be associated with a given business, but only one (1) business can be associated with a given person. Multivalued attributes can be restricted in a similar manner: An attribute mapping students to courses might be restricted to be [1:6], meaning that each student must be taking at least one course but no more than six courses. As detailed in Section 3.2, the IRIS data model permits the specification of several cardinality constraints on the same *n*-ary aggregation, thereby providing considerable expressive power.

Another prominent constraint is an *existence* constraint. This is related to a relational inclusion dependency and states that each entity of some type must occur in some aggregation. Consider the schema of Figure 3b, which represents the aggregation EMPLOYMENT. It makes no sense in this particular application for a business to exist in the database unless it participates in an EMPLOYMENT aggregation for at least one employee. To enforce this, we would say that there is an existence dependency between BUSINESS and EMPLOYMENT. It is also natural to place existence dependencies on attribute ranges.

The semantic modeling literature has also described constraints that are computed in nature: such constraints may involve schema components that are arbitrarily separated. These constraints are generally specified using a predicate describing properties of data taken from disparate parts of a schema. Such constraints in the World Traveler Database, for example, can state that for each business-traveler p, the city of p's employer is equal to the city where p lives or that the number of persons living in a given zip-code area is no greater than 10,000. Although several authors have suggested the usefullness of computed constraints in principle [Hammer and McLeod 1981; King and McLeod 1985a; Tsichritzis and Lochovsky 1982], no models in the literature support them formally.

There is a close relationship between integrity constraints and derived schema components. Both require that data associated with different parts of a schema be consistent according to some criteria. The essential difference is that an integrity

constraint does not extend the database with any new information, whereas derived data truly augment the database.

2.4 Manipulation Languages

Up to this point we have provided an overview of the data structuring mechanisms supported by typical semantic models. These capabilities would normally be supported by a *data definition language* associated with a specific model. No data model is complete without a corresponding *data manipulation language*, which allows the database user to create, update, and delete data that correspond to a give schema. In this section, we describe the general structure of a data manipulation language for the GSM and use it as a means of discussing the general nature of semantic data manipulation.

There are three fundamental capabilities that differentiate a semantic data manipulation language from a manipulation language for a traditional record-oriented model. First, the language must be able to query abstract types. Second, it must provide facilities for referencing and manipulating attributes. In this way, abstract, nonprintable information may be manipulated. Third, semantic manipulation languages often allow the user to manage derived data in the form of subtypes and functions constructed from existing (sub)types and functions. Thus, the specification of derived data is not reserved for the user of the data definition languages but may also be performed at run time. This blurs to some degree the traditional boundary between schema and data; the user's view of the world may now be extended dynamically with new information constructed from existing data. This provides a marked contrast with approaches taken in record-oriented models, in which the data definition and data manipulation languages are quite distinct.

Semantic data manipulation languages represent diverse programming language paradigms, but there are strong commonalities in terms of their functionality. Essentially, a semantic manipulation language typically takes the form of an extension to a language resembling a relational query language. Some semantic manipulation languages also include the flow-of-control and computational capabilities of general-purpose imperative programming languages. The GSM data manipulation language is a simple SEQUEL-like language.

Here is a query that lists the names of all linguists who speak three or more languages; it illustrates the basic capabilities of a semantic access language to manipulate types and functions:

```
for each X in LINGUIST
  such that LANGCOUNT(X) ≥ 3
print PNAME(X)
```

The next query prints any address such that more than one person resides at the given address:

```
for each X in ADDRESS
  such that for some Y in PERSON
  and for some Z in PERSON
    Y ≠ Z and
    ADDRESS(Y) = X and
      ADDRESS(Z) = X
print X.STREET, X.CITY, X.ZIP
```

Note that the "." notation is used to reference the various components of an aggregation. It is also true that if, for example, an address could have two components of the same type (e.g., two ZIPs), this notation would create an ambiguity. In general, it is necessary to be able to give names to the components of an aggregation and to reference them by those names, rather than by their types.

The following query illustrates the capability of a semantic language to manipulate derived information:

```
create subtype ROMANCE-LINGUIST of
    LINGUIST
  where SPEAKS includes French, Italian,
      Spanish, Portuguese, Rumanian,
      Sardinian
for each X in ROMANCE-LINGUIST print
    PNAME(X)
record ROMANCE-LINGUIST
```


The query creates a subtype, called ROMANCE-LINGUIST, of all linguists who speak French, Italian, Spanish, Portuguese, Rumanian, and Sardinian. Then the names of all romance linguists are printed, and the subtype is permanently recorded in the database schema. When a query specifies a derived subtype, it must be possible to name the subtype in order to reference it later. Again, we note that as a direct result of their rich modeling capabilities, semantic models require the creation of names that would not exist in a corresponding relational schema. Since such things as aggregations and subtypes may be created and referenced, they need names. This can be viewed as a limitation to the casual user who might feel that a semantic model causes a proliferation of names and therefore creates confusing schemas.

In the examples presented above, the output of the queries was a list of objects or values, not instances of semantic types. This is quite different from relational queries, which take relations as input and produce relations as output. As a result, in most semantic languages operations cannot be composed. Notably, the language FQL does not suffer from this limitation (see Section 3.5).

3. SURVEY

In this section we survey a number of semantic models. In particular, we discuss the first ten models (four horizontal groups) listed in Figure 12. We begin, in Section 3.1, with three models that are highly prominent in the literature. These are the Entity–Relationship (ER) Model, the Functional Data Model (FDM), and the Semantic Data Model (SDM). Then we briefly consider a number of other semantic models in Sections 3.2–3.4. Finally, in Section 3.5 we review the prominent semantic data manipulation languages.

The models of Sections 3.1 and 3.2 embody a number of explicit, distinct constructs in support of complex data modeling. Section 3.3 considers the binary models that offer only a minimal set of simple constructs, which are then used to build up more complex structures. In Section 3.4 we consider models that represent complex data by extending the relational model. The models in the last two horizontal groups of Figure 12 focus primarily on the research goals of encapsulating transaction facilities and theoretical investigations. These models are discussed in Section 4. (In this and all subsequent summary tables, a blank entry indicates that the specified feature is not present to the best of the authors' knowledge.)

The three prominent models and those discussed in Section 3.2 all explicitly support constructs for defining semantic databases. This approach has the advantage of providing a refined set of powerful modeling capabilities that the database designer and user may quickly comprehend. In contrast, the binary and relational extension models represent two very different philosophical approaches. The binary models take a building block approach in that they support only simple constructs that are then used to develop more complicated ones. This minimalist approach has the advantage of being more general: the models are very simple object-oriented ones that allow the designer to develop a wide variety of modeling constructs. In contrast, the relational extensions rely on underlying relational primitives to support higher level constructs. This approach has the advantage of being able to draw on a large body of knowledge concerning relational databases, which is useful in developing implementations and in enriching a system with integrity mechanisms, design methodologies, query optimization, and transaction specification facilities.

Figure 12 describes the various semantic models according to their structural and dynamic aspects. There are four main categories at the top of the figure: *References* indicates references to initial research on the models. *Philosophical Basis* classifies the models along three spectras: their primary research objectives, the nature of their underlying modeling primitives, and their general modeling philosophy. The research objective of each model is defined as providing a general-purpose semantic model, a basis for a structured design methodology, a programming language for

database applications, or a basis for theoretical investigation. The primitives in a given model are classified as being either explicit, minimalist, or relational extensions. The modeling philosophies are classified as being centered around attributes, aggregation, or more general type constructors, including aggregation (see Section 2.1).

The *Schema Components* category concerns the specific schema components supported by the individual models. We consider four capabilities: type constructors (including aggregation and grouping), attributes (including printable, object-valued, and multivalued attributes), ISA relationships, and derived schema components. In the figure, we indicate whether a capability is supported explicitly or implicitly, using combinations of other constructs (see Section 2). It is true, of course, that the lack of one of these capabilities or the choice of representing it implicitly rather than explicitly is not necessarily a deficiency; the underlying philosophy of a model is often simpler and cleaner if the number of constructs is kept minimal.

The *Dynamic Component* category concerns dynamic capabilities that are associated with each model. Specifically, literature references are provided to research that has been performed in the support of semantic query languages, data manipulation primitives, and transaction structuring. (Query languages and data manipulation primitives are discussed in Section 2.4, a survey of the prominent query languages is given in Section 3.5, and transaction structuring is discussed in Section 4.2.)

The general philosophy of semantic modeling has been applied to the development of special-purpose database models for applications such as very-large-scale integration (VLSI) design [Batory and Kim 1985]. Although these efforts are of significant interest and further validate the usefulness of semantic modeling, they are beyond the scope of this paper.

3.1 Prominent Models

In this section we consider three well-known models from the literature, which are considered by many to be the most influential semantic models. We see that, even though they are all explicit models, they vary greatly in their modeling capabilities and in their philosophical approach to database design. The first model constructs complex objects out of aggregations, which are visualized at the same level of abstraction as types. Consequently, such aggregations can be considered irrespective of the types they interrelate; that is, they are semantic entities on their own (see Section 2.1). In contrast, the two latter models use attributes as their primary method of constructing complex data; attributes are viewed as secondary to types and have no meaning at all without defining domain and range types. Thus, we see that the prominent models are representative of the two general modeling approaches described in Section 2.1.

3.1.1 The Entity–Relationship Model

The *Entity–Relationship* (ER) Model, proposed by Chen in 1976, is generally considered to be one of the first true semantic data models to appear in the literature, although the term "semantic" was not in use at the time. Schemas of this model have a natural graph-based representation and support the representation of abstract sets of entities, relationships between these entity sets, and attributes defined from both entity and relationship sets to printable values. Thus, an ER schema consists of types and relationships interconnecting these types, along with printable attributes of the types and relationships. Relationships can be restricted to 1:1, many:1, or many:many, and attributes can be restricted to 1:1.

Figure 13 shows a portion of the World Travelers schema using the standard ER notation. Rectangles represent *entity sets*, that is, abstract types, diamonds represent *relationships*, that is, aggregation, and ovals represent the ranges of attributes. The ER schema of Figure 13 illustrates a few differences between the original ER Model and other, later semantic models. The most fundamental difference is that ISA relationships are not represented. Recently, the ER Model has been extended to include ISA relationships [Baum et al. 1986; Teorey et al. 1986] (see Section 2.2.3).

Figure 12. Summary of prominent Semantic Data Models. Blanks indicate capability not supported to the authors' knowledge. References in this figure are beyond the scope of this paper.

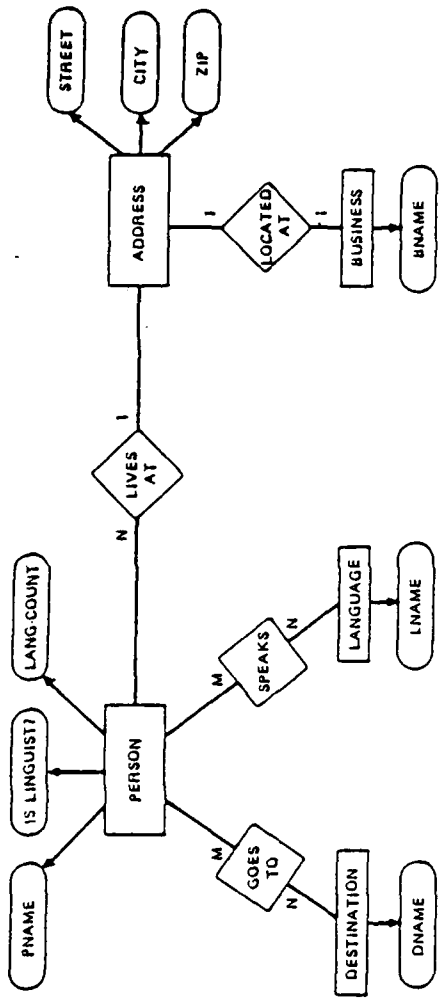


Figure 13. ER representation of part of the World Traveler schema.

Perhaps the most distinctive feature of the ER Model is the way in which it restricts the use of attributes and aggregation. Real-world attributes whose ranges are printable can be represented directly in the ER Model. For example, in Figure 13 ADDRESS has three printable attributes, STREET, CITY, and ZIP. On the other hand, real-world attributes that map to entity sets must be represented using relationships, as illustrated by the LIVES-AT relationship. (This relationship is shown to be many:1 because several persons might share the same address.) Because relationships are given names and, in a sense, viewed as entities themselves, it is straightforward in the ER Model to represent attributes of relationships, as illustrated in Figure 3b.

In the original ER Model, multivalued attributes also require the use of a relationship. This is because, as discussed above, attributes must be single valued. Thus, to represent the fact that a person may speak more than one language, the relationship SPEAKS is used to collect a number of languages into a set. The underlying philosophy is that an attribute is restricted to being a single fact about an entity, whereas a relationship can model the construction of more complex entities from other entities.

The ER Model was originally proposed [Chen 1976] as a schema design aid, permitting users to design schemas using a high-level object-based approach. The resulting ER schema would then be translated into either the relational or the network model. Within this framework, an ER schema is designed primarily for the purpose of articulating the overall data management objectives of an organization, but is not implemented per se. A detailed design methodology based on a generalization of the ER Model appears in Teorey et al. [1986].

In sum, the ER Model was the first semantic model centered around relationships, not attributes. It views the world as consisting of entities and relationships among entities. Both entities and relationships may have single-valued printable attributes. In the original ER Model ISA relationships are not supported, but recent researchers have proposed mechanisms for supporting ISA within the ER Model.

3.1.2 The Functional Data Model

The *Functional Data Model* (FDM) was introduced in 1976 [Kerschberg and Pacnec 376] and is recognized as the first semantic model centered around functional relationships, that is, attributes. Like the ER Model, a considerable amount of research has developed around FDM, and several other semantic models have adopted the attribute-based approach. Attributes in FDM can be either single- or multivalued and can be defined on domains that are Cartesian products of the atomic entity sets. FDM also supports ISA relationships. Significantly, the work of Shipman [1981] on FDM is among the first to include derived schema components as an integral part of a semantic model.

An informal graph-based representation of FDM schemas is introduced in Shipman [1981] and extended in Dayal and Hwang [1984] and elsewhere. An FDM schema corresponding to the World Traveler schema is shown in Figure 14. FDM connects objects directly with attributes without the use of intermediate constructs such as aggregation and grouping. This may be viewed as producing simpler schemas.

The data language DAPLEX [Shipman 1981] for this model was the first integrated data definition and access language formulated entirely in the high-level terms provided by an object-oriented semantic database model. DAPLEX was also the first database access language to give a prominent role to attributes, permitting their direct usage and also the use of their inverses and compositions. This and other semantic data access languages are discussed in Section 3.5.

FDM has spawned several research projects. It has been used to provide a unified interface to distributed heterogeneous databases in the Multibase project [Landers and Rosenberg 1982; Smith et al. 1981]. Integration of FDM schemas is studied in Dayal and Hwang [1984]. FDM also served as the basis for one of the

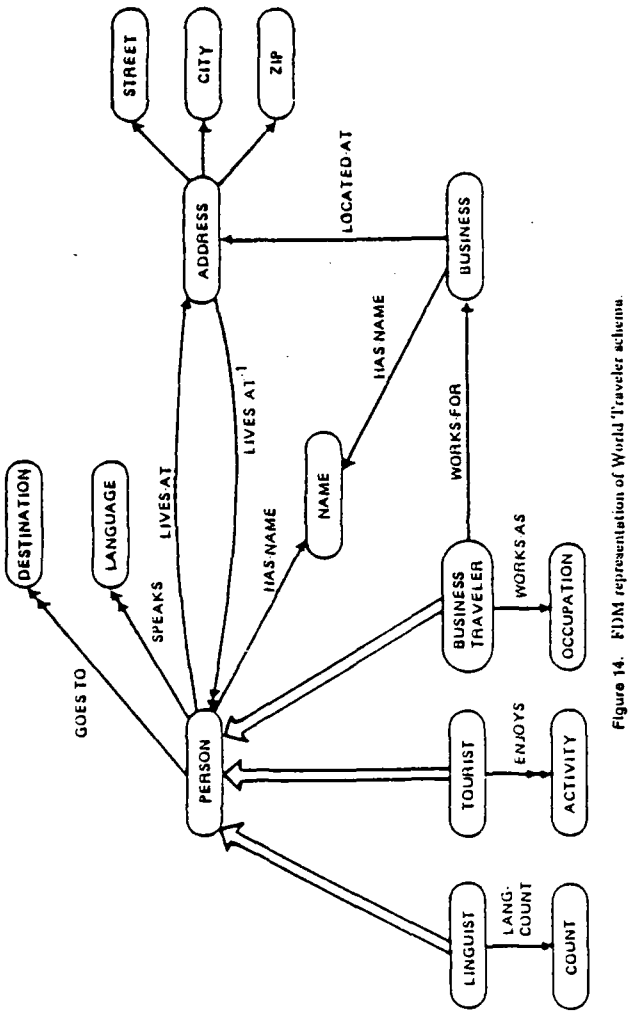


Figure 14. FDM representation of World Traveler schema.

original studies of update propagation in the context of semantic interconnections between data [Hecht and Kerschberg 1981] (see Section 4.4).

In summary, FDM was the first of a number of semantic models based on explicit representation of attributes with printables or objects as their ranges. It is a simple, elegant model with an easily understood visual representation. It gives prominence to atomic types and attributes, not to type constructors like aggregation and generalization. One of the major benefits of this model is the capacity to reference functions directly when manipulating properties of data.

3.1.3. The Semantic Database Model

The *Semantic Database Model* (SDM) [Hammer and McLeod 1981] was among the first published models to emphasize the use of the grouping constructor and the support of derived schema components. In particular, derived schema components permit *data relativism*, that is, multiple perspectives on the same underlying data set. SDM does not provide an explicit type constructor for aggregation, and in that sense it is attribute oriented (see Section 2); SDM does simulate aggregation with the attribute primitive.

SDM is unique in that it provides a rich set of primitives for specifying derived attributes and subtypes. For example, subtype relationships in SDM are broken into four categories: (i) those that are *attribute defined*, (ii) those defined by set operations (e.g., intersection) on existing types, (iii) those that serve as the range of some attribute, and (iv) those that are user specified (or *user controllable* in the terminology of Hammer and McLeod [1981]).

As an example of the second sort of subtype, we might form a subtype called RETIRED-TOURIST (retired people who travel), give it the same properties of TOURIST, and union it with TOURIST to give the new subtype ALL-TOURIST. An example of a subtype that exists explicitly to serve as the range of an attribute would be if TOURIST in the World Traveler schema were the range of an attribute, say HAS-BEEN-VISITED-BY, of a type COUNTRY. An example of the fourth sort of subtype would be SUSPICIOUS-TOURISTS, whose contents would be updated directly by the end user on the basis of personal criteria. Primitives for specifying derived attributes are also supported in SDM.

One of the predicates used in conjunction with the grouping construct is of particular interest because it provides expressive power of a structural nature. For example, in the World Traveler Database an *enumerated grouping class* called EMPLOYABLE-PERSON-TYPES can be defined to hold the already existing types LINGUIST and BUSINESS-TRAVELER. This is roughly equivalent to forming a metatype from a user-specified set of types. Attributes of this new type (e.g., number of elements, median income) are type attributes on the underlying types. Although this ability to view types as both sets and individual elements is found in only a few semantic data models (SAM* and TAXIS being other notable models), it is commonly supported by frame-based approaches to knowledge representation in AI [Fikes and Kehler 1985].

The richness of SDM as a schema specification language highlights the trade-off in semantic modeling between providing a small or large number of primitive data structuring constructs. In models with a small number of constructs, the representation of some data sets requires the artificial combination of these constructs; in a model with many constructs such as SDM, the designer is continually forced to choose from among a variety of ways of representing the same data. Thus, a model like FDM might seem stark compared with SDM, but for some users it might prove easier to learn and use.

SDM refines the notions of subtype and attribute by considering how they are defined. However, it does not support an explicit aggregation type constructor. Of our three prominent models, it appeared in the literature most recently. SDM, like FDM and unlike the ER Model, is centered

around attributes, but it is richer (and thus more complex) than either FDM or the ER Model.

3.2 Other Highly Structured Models

In this section we consider three other highly structured models, the Semantic Association Model (SAM*), the IFO Model, and the IRIS Model. SAM* and IRIS have been developed to support full-fledged database applications, whereas IFO was developed primarily for theoretical investigation. SAM* focuses largely on special forms of the aggregation construct, and both IFO and IRIS include both type constructors and attributes.

The *Semantic Association Model* (SAM*) [Su 1983], an extension of SAM [Su 1980], attempts to provide a set of constructs rich enough to exhaust the possible relationship types that might arise in both commercial and statistical applications. The model distinguishes different uses of some of the fundamental structural constructs of semantic models and in some cases provides them with different update semantics. As a result, the model supports a limited form of data relativism whereby a given construct might be viewed as having two or more different underlying structures within the same schema. The paper [Su 1983] presents a graph-based representation for SAM* schemas and also suggests an approach to implementing SAM* based on data structures called *G-relations*, which are closely related to non-first-normal-form relations [Abiteboul and Bidoit 1984; Fischer and Thomas 1983; Jaeschke and Schek 1982; Makinouchi 1977]. Schema definition and data manipulation languages for SAM* are under development [Su 1986].

The basis for SAM* schemas is provided by what are called *atomic concepts* in Su [1983]. These include integer, real, character-string, and Boolean types, as well as structured programming language data types constructed from these, including vectors, arrays, and ordered and unordered sets, time, time series, text, and G-relations.

In SAM* physical or abstract objects are represented using *nonatomic concepts*. Seven kinds of *associations* or type constructors for building these concepts are used. The different associations have somewhat specific intended uses, and operations especially tailored to them.

We first consider four associations typical of most database applications. First, *membership association* permits the specification of the domain of a type or an attribute range. For example, a membership association could be used to specify that the type CITY has individual members Paris, Beijing, and so on. The SAM* representation of this is shown in Figure 15a. (The M in the root node indicates that it is a membership-association.) The next two kinds of association are closely related to aggregation. The first of these is called *aggregation association* and is illustrated in Figure 15b. This kind of association is typically used to provide a description of an entity and its attributes. In the figure, PNAME is indicated to be a key. In terms of the GSM, SAM*'s *interaction association* also has the structure of aggregation. In SAM*, this is typically used to represent abstract concepts based on more concrete concepts. For example, the ENROLLMENT type (consisting of pairs of courses and students) can be modeled as an interaction association. Cardinality conditions can be specified for these associations. In some cases a given concept can be viewed in a SAM* schema as both an aggregation association and an interaction association. In this way different users can view the same schema in different ways. Finally, *generalization associations* are used to model ISA relationships. A generalization association forms the union of its underlying types. Constraints such as disjointness can be specified.

The remaining three associations focus on aggregate properties of sets and are especially suited for statistical applications. One of these is *cross-product association*. This form of association is again an aggregation in the terminology of GSM but typically holds *all* of the Cartesian product of the underlying types instead of a subset of

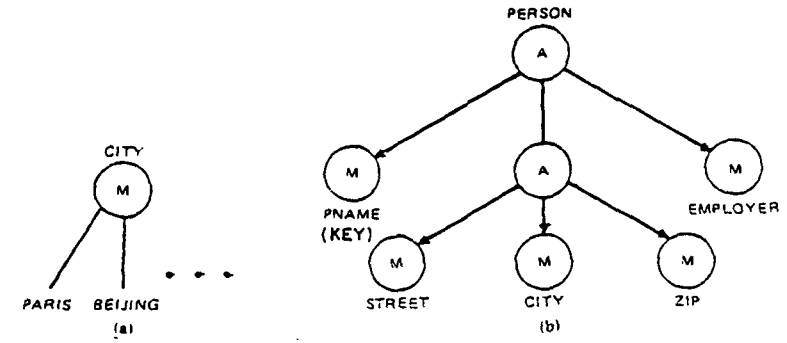


Figure 15. Membership and aggregation associations in SAM*.

that product. This association might be used to identify a variety of categories; for example, in a statistical database on populations it may be useful to consider categories formed from triples of AGE-RANGES, RACE, and SEX. These triples would then serve as the domain of functions describing statistical features of the different population groups they delimit. A *composition association* is used to hold a vector of sets. For example, a composition association for CURRENT-FLEET might be a triple, with three coordinates for the current sets of cars, trucks, and boats, respectively, that a business owns. In an instance, a composition association node will hold exactly one such tuple. Composition associations can participate in aggregations. Finally, *summarization association* is used to specify attributes for both cross-product associations and composition associations. These attributes typically hold statistical values and are thus a form of derived data. In the case of cross-product associations the attributes will be attributes in the sense of GSM; in the case of composition associations they will be type attributes.

To summarize, the seven kinds of associations used in SAM* have overlapping semantics but are distinguished by their associated update semantics and the constraints permitted. Although not discussed here, a variety of local restrictions is placed on how the constructs can be combined with each other, thereby ensuring that SAM* schemas are meaningful.

SAM* has recently been applied to the area of manufacturing data [Su 1986]. In particular, SAM* has been shown to be useful in representing the semantics of such complex data types as temporal data, recursively structured data, replicated data, and versions.

The *IFO Model* [Abiteboul and Hull 1987] was developed to provide a theoretical framework for studying the structural aspects of semantic data models. The model incorporates attributes and type constructors for aggregation and grouping at a fundamental level and distinguishes between two kinds of ISA relationships. The model is used in Abiteboul and Hull [1987] to characterize the propagation of simple updates in the presence of semantic model relationships and to analyze formally the interplay between constructed types and ISA relationships.

In many respects the IFO Model is similar to (and inspired) the structural portion of the GSM; a fundamental difference concerns how IFO combines the semantic constructs in forming schemas. The basic building block of an IFO schema is called a *fragment*. Fragments are used as abstraction mechanisms for representing an object type along with its internal structure and its attributes. Figure 16, which shows the IFO representation of a portion of the

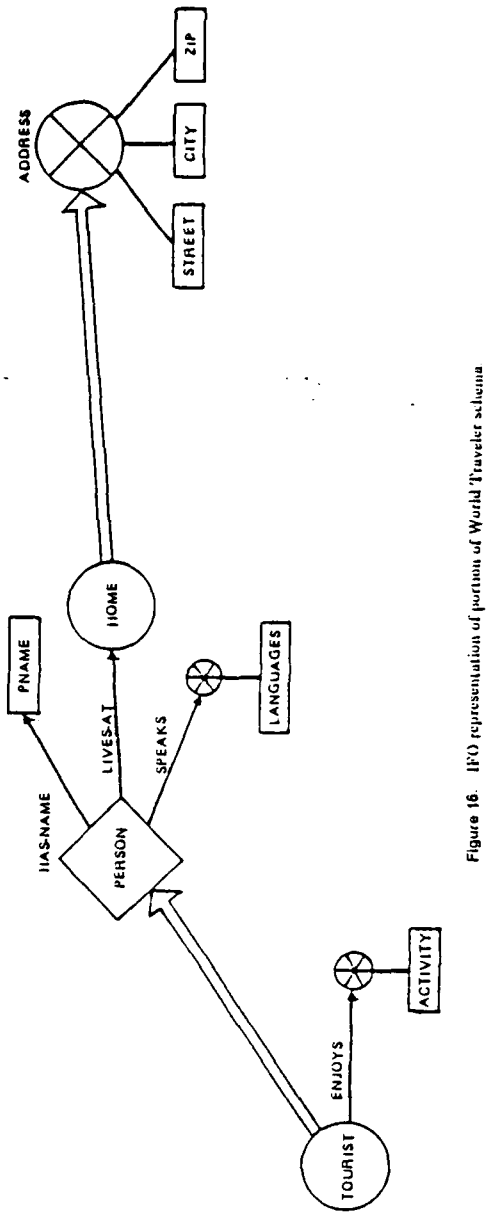


Figure 16. IFO representation of portion of World Traveler schema

World Traveler schema, contains three fragments: one for PERSON, one for TOURIST, and one for ADDRESS. The fragment for PERSON illustrates how IFO clusters information about a type and its attributes. In particular, the HOME node is used as a place holder for the range of LIVES-AT. This node is shown as a *free* (circle) node and is restricted by a specialization edge to take its values from the type ADDRESS. Free nodes are used in IFO to indicate that the type of objects populating it is determined through ISA relationships by another part of the schema. In IFO, nodes such as HOME that occur as attribute ranges cannot be used in the same manner as atomic type nodes or the roots of constructed types. The specialization edge from HOME to ADDRESS enforces a form of referential constraint; this is related to but somewhat different from the subtype constraint represented by the ISA edge from TOURIST to PERSON. In IFO the leaves of constructed types are also represented as free types that are restricted using referential constraint ISA edges.

The use of fragments in IFO highlights some of the similarities between semantic models and frame-based approaches to knowledge representation such as KL-ONE [Brachman and Schmolze 1985]. A fragment in IFO corresponds loosely to a frame. Fragments provide a natural way of representing *nested* or context-dependent attributes. To illustrate, consider the set-valued attribute SPEAKS of the World Traveler schema. In IFO this could be augmented with a nested attribute WITH-PROFICIENCY, which would specify the proficiency. For example, if Mary spoke French and Chinese, this nested attribute might state that she speaks French with proficiency of 2 and Chinese with a proficiency of 3. Furthermore, the IFO model distinguishes between two kinds of ISA relationships, essentially as described in Section 2.2.3.

The *IRIS Model* [Derrett et al. 1985] was introduced recently, and a number of research projects using the model have been undertaken. The model is based primarily on object types, specialization, multivalued attributes, and some forms of derived schema components. An initial prototype version of IRIS [Derrett et al. 1985] will include a nonprocedural language for queries and specifying derived data, as well as schema definition capabilities. The use of the IRIS Model as the basis for software specification has been investigated [Lyngbaek and Kent 1986], and a theoretical investigation of the model has been initiated [Lyngbaek and Vianu 1987]. We focus on the support of derived data in the model and on the constraints used.

The basic building blocks of the structural portion of IRIS are readily described in terms of the GSM. IRIS supports both *literal* (printable) and *nonliteral* (abstract) object types. These types participate in a directed acyclic graph of ISA relationships that has the unique type OBJECT at its top. Objects may also be related through (typically multivalued) attributes, whose domains and ranges may be types or cross products of types. Aggregations are modeled using a single-valued attribute from a cross product of types into the Boolean type.

IRIS uses derived schema components to support a form of data relativism whereby the same data can be viewed structurally from more than one perspective. In particular, IRIS permits the derivation of several attributes from a single *base predicate*, that is, aggregation. Figure 17 illustrates this point with a simple example. The base predicate shown in Figure 17a specifies 4-tuples describing enrollments. We assume here that each COURSE is offered in several different LECTUREs, that STUDENTs take a given lecture of a course, and that students receive a GRADE. In Figure 17b, this information is viewed using the attribute COURSE-STUDENT-STATUS, which maps each course–student pair into the lecture the student is taking and the grade received. IRIS permits the specification of this attribute as being derived from the base predicate. An important underlying principle of IRIS is highlighted by this example: The attributes ENROLLMENT and COURSE-STUDENT-STATUS are viewed as independent of any particular underlying type. In general,

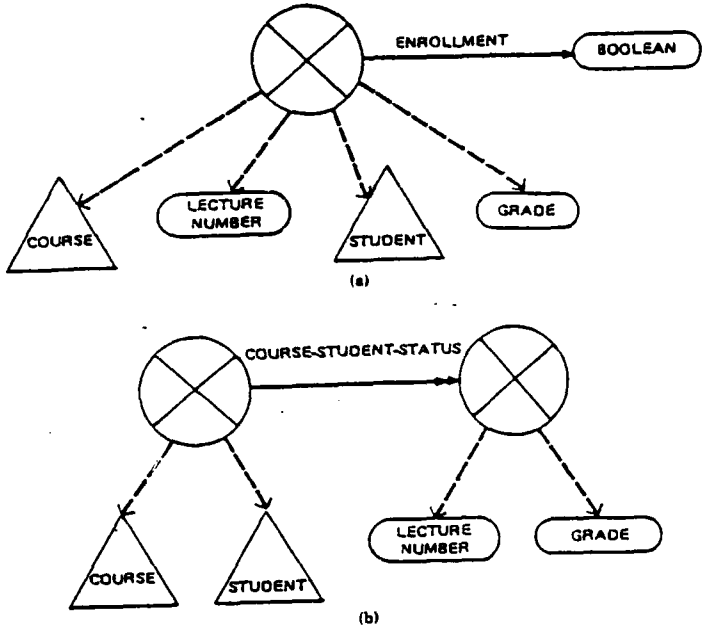


Figure 17. Two views of aggregation supported in IRIS.

several different attributes can be derived from a given base predicate.

IRIS supports a powerful kind of cardinality constraint called an *object participation* constraint. On the base predicate of Figure 17a, the constraint STUDENT[1, 5] would indicate that each student must be enrolled in at least one course–lecture pair and cannot be enrolled in more than 5. In Lyngbaek and Vianu [1987] these constraints are extended to attributes to address more than one coordinate at a time. For example, COURSE-STUDENT[0, 1] is used to state that each course–student pair can appear 0 or 1 time in the base predicate. This implies that the attribute of Figure 17b is single valued. An analysis of these constraints is presented in Lyngbaek and Vianu [1987]. Translation of IRIS schemas into relational schemas is also presented there.

3.3 Binary Models

In this section, we consider a representative of the family of "binary" models, which attempts to supply a small, universal set of constructs that are used to build more powerful structures. These models are thus minimalist in the sense that they require the database designer to understand fewer constructs.

The *Semantic Binary Data Model* (SBDM) [Abrial 1974] is representative of this family of models [Bracchi et al. 1976; Deheneffe et al. 1974; Hainaut and Lecharlier 1974; Senko 1975], all of which represent data using two constructs: entity sets and binary relations. As indicated in Figure 18, schemas of these models typically consist of labeled nodes for entity sets and labeled arcs corresponding to binary relationships between them. The primary

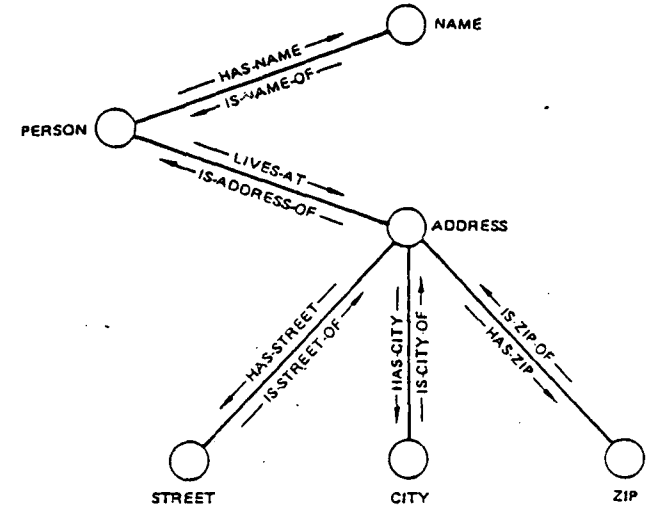


Figure 18. SBDM representation of part of World Traveler schema.

contribution of these models was to provide an early vehicle within which a number of fundamental types of data relationships could be articulated and studied.

In the SBDM, each binary relation is actually viewed as an inverse pair of possibly multivalued functions. For example, the binary relation connecting PERSON with ADDRESS in Figure 18 is viewed as consisting of the (single-valued) function LIVES-AT and the (multivalued) function IS-ADDRESS-OF. A data definition language and data manipulation language for the SBDM are defined in Abrial [1974]. The defined database transactions can be stored within the database; in this respect the SBDM follows INGRES [Stonebraker et al. 1976], in that the data dictionary access language is the data manipulation language applied to a certain part of the database. Recent work with these models includes the use of the SBDM as the basis for a formal schema design methodology [Dardailler et al. 1985], and the use of an extended binary model that incorporates ISA relationships and local constraints on relationships (e.g., 1:1 or N:1) [Rishe 1985, 1986].

As just described, the SBDM is closely related to the GSM in representational power. A major difference is that the type constructors of GSM must be simulated in SBDM. For example, the aggregation ADDRESS of the World Traveler schema is modeled as an abstract type with three binary relationships in the SBDM schema of Figure 18. As with FDM, a constraint must be added to ensure that each STREET, CITY, and ZIP tuple is unique. Also, the SBDM as described in Abrial [1974] does not support ISA relationships, although such relationships can essentially be represented within the framework.

3.4 Relational Extensions

The ER Model, FDM, SDM, and the other explicit models are similar in that they all take the approach of supplying a handful of distinct constructs that together are designed to serve the vast majority of modeling situations arising in typical database

application environments. They differ largely in the approach taken to interrelate data (type constructors versus attributes) and in the number of constructs supported. In this section we consider a very different approach taken by some researchers. We examine several models that support complex data as an extension to the well-known relational model. In these models the designer and user of a database view data from the relational perspective but are given mechanisms built out of the relational model to construct semantic schemas. A benefit of this approach is that these models may draw on the large body of knowledge known about the relational model, including query optimization, implementation strategies, and query language formulation. A potential drawback is that each of these models, like the relational model, imposes a level of indirection owing to the representation of objects and relationships based on records and identifier comparisons. By some users this might be viewed as tedius and inelegant.

The *Structural Model* [Wiederhold and El-Masri 1980] is a relatively simple extension of the relational model, which was introduced primarily as a tool for designing and integrating schemas for the record-oriented models. In this model, data are stored in relations, and five types of relation are distinguished. First, *primary entity* relations are used to store sets of tuples that closely correspond to classes of entities in the world. These relations store identifiers for these entities, along with single-valued attributes defined on them. In general, primary entity relations will not be affected by updates on other parts of the schema. *Referenced entity* relations, on the other hand, are used for entity sets that serve primarily as the range of attributes defined on the primary entity types. *Nest* relations are used for holding many-valued attributes, and *lexicon* relations are used to hold 1:1 correspondences between different names for the same object (e.g., person-name and Social Security number). Finally, *association* relations serve the same role as relationships in the ER Model and are used to model many:many relationships between primary entity types. Additional semantics are incorporated into the model by restricting how these different types of relations can reference each other (e.g., a referenced entity relation must be referenced by at least one other relation). In sum, the Structural Model uses relations to simulate an object-oriented approach that incorporates aggregation and single- and multivalued attributes in a fairly direct manner. On the other hand, ISA relationships are not incorporated as directly. An interesting application of the Structural Model is described in Brown and Parker [1983]. This paper introduces a graph-based representation of Structural Model schemas and describes a methodology for simplifying them.

RM/T [Codd 1979] is Codd's extension of the relational model. As in the Structural Model, various kinds of relations for representing different semantic modeling constructs are distinguished, and update semantics are specified for them. In RM/T, abstract objects are represented by *permanent surrogates*, and each type has an associated one-column *E-relation* that holds the surrogates of the objects currently populating that type. Although users can request that surrogates be created or deleted, they can never explicitly reference or view them. In this way, RM/T closely follows the object-oriented spirit of most semantic models. Single- and multivalued attributes are stored in relations using surrogates and printable values. Two forms of aggregation are supported: The so-called *associative entities* are aggregation objects that are assigned a new surrogate. The *nonentity associations* are aggregations for which no surrogates are assigned; these aggregations cannot have multivalued attributes, nor can they participate in ISA relationships. Grouping types based on attribute values (see Section 2.3.2) are called *cover aggregations* in RM/T. The model also provides explicit constructs for representing precedence relationships between entities that have time-valued attributes.

RM/T supports two types of ISA relationships. *Unconditional generalization* is

essentially the notion of ISA used in the GSM: each object in a subtype of an unconditional generalization must be a member of the supertype. *Alternative* (or conditional) *generalization* is used to form subsets of a union of types. For example, the type CUSTOMER might be defined as an alternative generalization of PERSON, BUSINESS, and PARTNERSHIP. In RM/T, this means that each customer must be either a person, a business, or a partnership, but that the set of customers does not have to contain all persons, businesses, and partnerships. Note that in some models, the type CUSTOMER could be modeled by first forming a supertype LEGAL-ENTITY of the three base types and then defining CUSTOMER as a subtype of that.

GEM [Tsur and Zaniolo 1984; Zaniolo 1983] is a relational extension that can also be viewed as an extension of the ER Model. In particular, this model supports entities and relationships, as well as subtyping and nonatomic attribute ranges. An unusual aspect of GEM is that it was developed as an experiment in supporting a semantic data language by extending a relational query language (see Section 3.5). This is in contradistinction to investigations that extend the relational model itself.

3.5 Access Languages

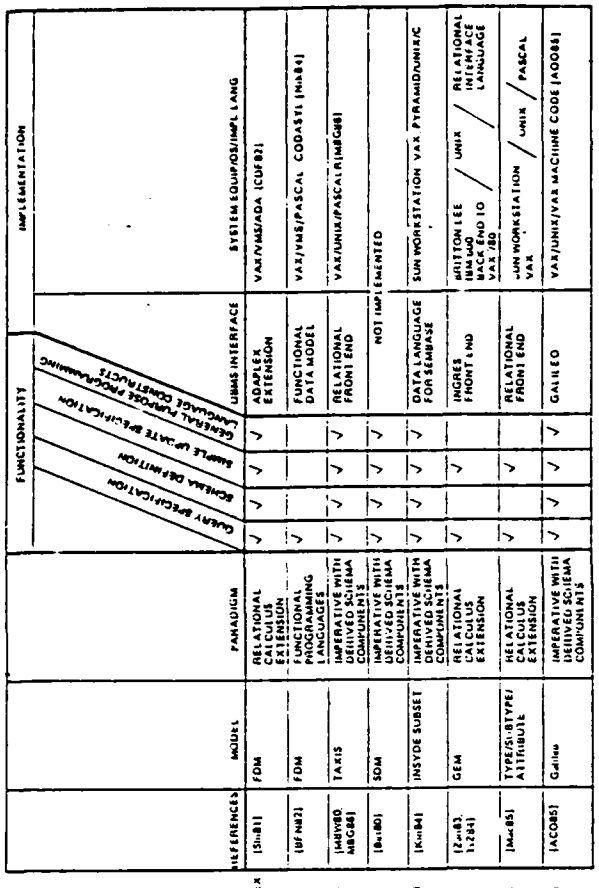
We conclude our survey of semantic models by examining various access languages that have appeared in the literature and that support semantic modeling constructs. We include these languages in this section (and not in Section 4, which discusses research directions of semantic modeling) because several of them were defined concurrently and independently of the various semantic models. Indeed, numerous researchers have taken the approach of viewing data modeling and data manipulation as an integrated mechanism. In this section we illustrate the general capabilities of semantic access languages using a language similar in form to DAPLEX [Shipman 1981] and Semdal [King 1984] and briefly survey the prominent languages in the literature. Figure 19 gives a brief survey of these languages. In this section we do not consider data manipulation for deeply nested constructed types; theoretical approaches to this problem are discussed in Section 4.4.

Three of the access languages, DAPLEX, GEM, and ARIEL, are extensions of the relational calculus [Date 1981; Ullman 1982] designed to encompass standard relational as well as semantic data structures. FQL is unique because it is based on the paradigm of functional programming (not to be confused with the Functional Data Model, which it supports). Unlike most database query languages, FQL does not support update specification or schema definition. Finally, TAXIS, DIAL, Semdal, and Galileo are imperative languages, with philosphical similarities to typical Pascal-like languages, including standard flow of control facilities and arithmetic capabilities.

DAPLEX supports the Functional Data Model. Like the other languages in this class, the query specification portion of this language contains syntactically elegant renditions of most of the basic elements of the first-order predicate calculus and is thus fundamentally nonprocedural. DAPLEX also supports the direct mention of attributes, their inverses, and their compositions and thus permits queries to have a somewhat navigational flavor. It also supports the specification of aggregate values such as averages, of orderings and similar properties of entity sets, and of database updates. It also supports the definition of schema components, including derived data.

FQL finds its roots in the work of Backus [1978] on functional programming languages. This approach offers several advantages. In particular, the functional approach reduces the use of secondary storage, simplifies the interface needed to provide outside programs with database access, and also typically enjoys an implementation that is quite compact [Buneman et al. 1982].

A query in FQL is formed by composing one or more functions, which may themselves be formed using transformations such as inverse or *, which turns a single-valued function into its analog that maps sets to sets. Another important operator is *restriction*, denoted using a |, which acts like a filter on a list of values. To illustrate



	REFERENCES	MODEL	PARADIGM	QUERY SPECIFICATION	SCHEMA DEFINITION	SIMPLE UPDATE SPECIFICATION	GENERAL PURPOSE PROGRAMMING LANGUAGE CONSTRUCTS	USMS INTERFACE	SYSTEM EQUIP/OS/IMPL LANG
DAPLEX	[Shi81]	FDM	RELATIONAL CALCULUS EXTENSION	✓	✓	✓	✓	ADAPLEX EXTENSION	VAX/VMS/ADA [CUF82]
FDL	[BFN82]	FDM	FUNCTIONAL PROGRAMMING LANGUAGES	✓				FUNCTIONAL DATA MODEL	VAX/VMS/PASCAL CODASYL [Mos84]
TAXIS	[MBW80, MBG86]	TAXIS	IMPERATIVE WITH DERIVED SCHEMA COMPONENTS	✓	✓	✓	✓	RELATIONAL FRONT END	VAX/UNIX/PASCAL [IMG86]
DIAL	[Bro80]	SDM	IMPERATIVE WITH DERIVED SCHEMA COMPONENTS	✓	✓	✓	✓	NOT IMPLEMENTED	
Semdal	[Kun84]	INSYDE SUBSET	IMPERATIVE WITH DERIVED SCHEMA COMPONENTS	✓	✓	✓	✓	DATA LANGUAGE FOR SEMBASE	SUN WORKSTATION VAX PYRAMID/UNIX/C
GEM	[Zan83, Lyn84]	GEM	RELATIONAL CALCULUS EXTENSION	✓		✓		INGRES FRONT END	MOTTO/LEE IBM 4340 BACK END TO VAX /80 / UNIX / RELATIONAL INTERFACE LANGUAGE
Adiel	[Mac85]	TYPE/SUBTYPE/ ATTRIBUTE	RELATIONAL CALCULUS EXTENSION	✓		✓		RELATIONAL FRONT END	SUN WORKSTATION VAX / UNIX / PASCAL
Galileo	[ACO85]	Galileo	IMPERATIVE WITH DERIVED SCHEMA COMPONENTS	✓	✓	✓	✓	GALILEO	VAX/UNIX/VAX MACHINE CODE [ADO86]

Figure 18. Data languages based on semantic models. Blanks indicate capability not supported to authors' knowledge. References in this figure are keyed to the reference list.

the approach, we present an FQL-like query that corresponds to the first query of Section 2.4. In this query a list of all linguists is formed using !LINGUIST. This list is filtered by an operator, which first forms ordered pairs with first coordinate the language count associated with a given linguist, and second coordinate the number 3, and then determines whether this pair stands in the ≥ relation. At this point, then, a list of linguists who speak at least three languages has been formed. This functional is now composed with *PNAME (which is the natural extension of the attribute PNAME to lists of persons):

!LINGUIST ∘ ⌊ ([LANG-COUNT,3] ∘ ≥)
∘ *PNAME

As discussed above, FQL does not support database updates or schema definition, and it is not clear how such capabilities could be incorporated into the language in a natural fashion.

TAXIS, Galileo, DIAL, and Semdal are designed to act as both query and application languages and are thus called *database programming languages*. The advantage of this approach is a cleaner integration of semantic database operations into application programs. The goal is to support the flexibility of general-purpose programming language constructs in constructing database transactions. In a database programming language, the user can write application programs that use semantic data types as program data types—and count on them to persist until the next use of the database, without having consciously to separate database archival tasks from general programming tasks. In this way, the user may extend the conceptual benefits of manipulating complex objects and ISA hierarchies into nondatabase manipulations. TAXIS and Galileo are discussed further in Section 4.2 below; an excellent survey of these and other programming languages can be found in Atkinson and Buneman [1987].

4. FROM IMPLEMENTATIONS TO THEORETICAL ANALYSIS

In this section we examine a number of research projects that have evolved around semantic modeling. These projects, instead of focusing on constructing new modeling techniques, have studied and applied semantic modeling to such areas as the implementation of physical access methods of semantic databases, the construction of mechanisms for the specification of database dynamics the design of graphical interfaces to databases, and the theoretical analysis of database issues.

After semantic models and data languages based on semantic models had been in the research literature for a few years, some researchers began to wonder about the possibility of constructing full, efficient DBMSs based on these constructs. In Section 4.1 we discuss various implementation issues unique to semantic models and survey five implementations described in the literature.

In this article we have focused primarily on the static aspects of semantic models and discussed how they are used to structure and modularize database schemas. Some researchers have extended the area by developing mechanisms for structuring and modularizing the dynamic aspects of these models, with a primary focus on database transactions. In Section 4.2 we discuss two approaches researchers have taken in this area and survey the TAXIS, Galileo, SHM+, and INSYDE data models.

Section 4.3 examines what is perhaps the newest application of semantic modeling. A number of researchers have discovered that semantic models make useful mediums for providing interactive database interfaces. We examine several experimental graphics-based interfaces that use this approach.

Finally, Section 4.4 discusses theoretical investigations of concepts arising in semantic modeling. These include the development of formal query languages for constructed types, characterizations of update propagation, and other topics. This section also surveys the Format and Logical Data Models.

4.1 Systems

Two fundamental issues arise in the construction of full, efficient DBMSs based on the paradigm of semantic modeling. On the one hand, semantic models contain unusual constructs that force the physical designer

of a database to develop unique implementation constructs—abstract objects, subtype hierarchies, and derived data, in particular, introduce new issues in database physical design. On the other hand, like hierarchical and network databases, semantic schemas suggest expected access patterns to the designer. The designer of the physical implementation may capitalize on this information in selecting appropriate data structures.

In studying this area researchers have concentrated on two broad strategies: the use of existing data management capabilities (like relational systems and persistent programming languages) and the development of special-purpose, highly efficient access mechanisms. These efforts vary in the depth of their implementations and in the goals of the projects themselves. Figure 20 describes five sample systems in chronological order of their presentation in the literature. In the following these systems are described in some detail.

Three of the systems use existing tools to perform low-level data management: EFDM uses persistent ALGOL, GEM is built on top of Britton Lee's IDM500 relational database machine, and TAXIS is implemented in PASCALR. In both GEM and TAXIS each type is mapped to a relation. For greater efficiency, ADAPLEX and Sembase directly implement storage and access mechanisms to support semantic constructs. Objects are represented as variable-length records, with a field for each attribute connection. Each such field contains a reference to an object or a set of objects. These two implementations use conventional access methods, like B-trees, to manage sets of related objects (e.g., subtypes).

The five systems differ fundamentally in their intent as research projects. TAXIS is not intended to be a DBMS; rather, it is designed to support a programming language that encompasses data management facilities based on a semantic model. This language is described further in Section 4.2. ADAPLEX is an experiment in efficiently implementing a semantic database using general-purpose operating system files and in embedding a semantic database language (DAPLEX) in the programming language Ada. GEM is an attempt to support a semantic front end that is compatible with existing relational DBMSs and to assess the usefulness of relational database machines in supporting semantic models. The EFDM system concentrates on using a persistent language as a database implementation tool.

Sembase is an experiment in producing an integrated semantic language and graphical interface (see Section 4.3) and in providing efficient maintenance of derived data. A novel aspect of the Sembase implementation is the homogeneous treatment of records corresponding to objects of various types. All object records are stored in a heap on disk and accessed using a system of B-trees. In this way Sembase can adapt to predominantly used access patterns. Sembase also provides very efficient means of keeping derived subtypes and some forms of derived attributes up to date so that only a minimal amount of processing is required when a user requests a piece of derived information. To do this, Sembase maintains complex dependency information at the data level. For example, in the World Traveler Database, if a person speaks one language and learns another, the system will know that the given traveler must be reevaluated vis-à-vis the derivation rule for LINGUIST; the subtype as a whole will not be reevaluated. A recent extension of Sembase called CACTIS [Hudson and King 1986, 1987] supports the elegant and efficient maintenance of a much wider class of derived attributes.

An extension of ADAPLEX, called the Distributed Data Manager [Chan et al. 1983], supports a distributed semantic DBMS. This system uses the Functional Data Model and demonstrates that it is possible to implement a distributed semantic DBMS efficiently. Further, a semantic model is shown to be useful in creating more efficient distributed systems by making use of the extra semantics supplied in a schema when deciding how to distribute data. In particular, the Distributed Data Manager uses "fragment groups" as a means of localizing interobject references. Essentially, derived subtyping is used as a means of creating a small level of granularity to describe how different types of data

	REFERENCES	MODEL	IMPLEMENTATION STRATEGY	SYSTEM MACHINE/OS/IMPLEMENTATION LANG	ASSOCIATED SYSTEM COMPONENTS
TAXIS	[MBG86]	TAXIS	RELATIONAL FRONT END, TYPE PER RELATION	VAX/UNIX/PASCALR	FULL PROGRAMMING LANGUAGE CAPABILITY [MBG86]
EFDM	[AsK83, KuB83]	FDM	PERSISTENT ALGOL FILE PER TYPE OR SUBTYPE	VAX/VMS/PERSISTENT ALGOL	QUERY LANGUAGE [ShuB1]
GEM	[LuZ84]	GEM	RELATIONAL FRONT END, TYPE PER RELATION	BRITTON LEE IDM 500 / UNIX / RELATIONAL INTERFACE LANGUAGE BACK END TO VAX 780	QUERY LANGUAGE [Zan83]
ADAPLEX	[CDF82]	FDM	VMS FILE PER TYPE OR SUBTYPE	VAX/VMS/ADA	EMBEDDED IN HOST PROGRAMMING LANGUAGE [SFL81] DISTRIBUTED IMPLEMENTATION [CDF82]
Sembase	[Kin84, FKM85]	INSYDE	EXTERNAL NON UNIFORM RECORD PER OBJECT	SUN WORKSTATION / UNIX/C VAX, PYRAMID	QUERY LANGUAGE [Kin84] GRAPHIC INTERFACE [KiM84]

Figure 20. Systems based on semantic models. Blanks indicate capability not supported to authors' knowledge. Keys to references in this figure precede entries in reference list.

relate. Thus, for example, travelers may be grouped according to where they have been and then localized near the appropriate site information. In this way fragment groups are used as the basis of distribution and replication.

4.2 Dynamics

Until recently, most semantic modeling research concentrated on specifying the structural aspects of constructed types and the relationships between them, not on the behavioral components of semantic models. In Section 2 we discussed the dynamic components of semantic models for schema specification and data manipulation at a fundamental level. In this section we consider the broader issue of providing facilities for structuring database manipulation primitives into transactions.

The work on semantic transaction specification stems from early work on the design of programming languages with embedded database access mechanisms (e.g., [Rowe and Schoens 1979; Wasserman 1979]) and the work of Abrial [1974] on data semantics in the context of the SBDM. Two philosophies emerge in the four models discussed here. TAXIS and Galileo synthesize semantic modeling constructs with control and typing mechanisms from imperative programming languages. In contrast, SHM+ and INSYDE develop control mechanisms that closely follow the structure of semantic schemas.

TAXIS [Mylopoulous et al. 1980] is typically viewed as a programming language for data-intensive applications that incorporates several of the fundamental principles of data representation found in the semantic database literature. In particular, TAXIS is recognized as one of the first systems to merge semantic data modeling concepts, including attributes and ISA relationships, with more general programming language facilities such as abstract data types and exception handling. TAXIS provides tools for modularizing the specification of database transactions and can support a wide class of interactive data management applications.

To support the semantic data modeling concepts, TAXIS uses an extended form of the relational model (see Section 3). TAXIS incorporates most of the semantics associated with data in the specification of the operations associated with those data.

A more recently developed language, Galileo [Albano et al. 1985], is similar to TAXIS in that it incorporates semantic data model mechanisms within a strongly typed programming language. Galileo introduces a general family of type constructors for both database and nondatabase types and then uses them to support aggregation, attributes, and ISA relationships between database types. An interesting contribution of Galileo is the concept of *environments*, which are used as an abstraction mechanism in the support of modularization. Environments allow data operations to be incorporated into the database schema. Also, environments allow schemas to be built incrementally by giving the database user a mechanism for specifying, in a controlled fashion, exactly how new operators and schema components should interact with the existing schema.

The *Extended Semantic Hierarchy Model* (SHM+) [Brodie and Ridjanovic 1984] has highly structured static and dynamic components. The model for static data representation used in SHM+ is based on aggregation, grouping, and subtypes and represents attribute relationships using aggregation and keys. Static schemas in the model have a very hierarchical and modular flavor. In SHM+, behavioral components form an integral part of the structural building blocks. For example, a primitive for iterative application of an operation is directly associated with the grouping construct. As a result, it is very easy to specify involved database updates as highly structured transactions [Brodie and Ridjanovic 1984]. A comprehensive methodology for incremental design of the static and dynamic components of schemas is developed in Brodie and Ridjanovic [1984].

The *INSYDE Model* [King and McLeod 1985a], which is derived from the Event Model [King and McLeod 1982, 1984] is a semantic database model that combines static as well as dynamic primitives at a fundamental level. The static portion of the model is essentially a rich subset of SDM, and the dynamic component includes

language constructs to support data manipulation as well as schema design and evolution. The INSYDE Model provides a unified support system for the entire life cycle of a database, providing tools and abstraction mechanisms for the design of the static schema, the design of database transactions, and the support of schema evolution within an integrated framework. In particular, it includes a prescriptive, stepwise methodology for logical database design, the purpose of which is to guide a designer through both the specification and maintenance of a semantically expressive schema. This methodology gives a primary role to transactions: The expected transactions are a fundamental part of the initial functional specification of a system and are used to drive much of the refinement and implementation process. Use of the INSYDE Model is particularly appropriate in the design of Office Information Systems [King and McLeod 1985a].

4.3 Graphical Interfaces

Several graphical interfaces to databases based on semantic models have been developed. This is due largely to the fact that semantic models are conducive to visual representations. In fact, as we have seen, the data definition languages of some semantic models are already graphical in nature. The development of these interfaces has also been encouraged by the advent of workstation technology, which provides relatively inexpensive bit-mapped displays and pointing devices (such as mice). Semantic models are an obvious choice for capitalizing on the ability of bit-mapped workstations to display two-dimensional images. In this section we examine a few noteworthy attempts at using a semantic model as the basis of an interactive database interface.

Figure 21 surveys six experimental database interfaces based on semantic models. In Figure 21 we construct a taxonomy of these systems along three central axes: functionality, implementation environment, and the extent to which graphics are used. With each system, we also list the data model the interface supports. The six systems are listed chronologically, in terms of their presentation in the literature. We see that they tend to support more and more complex data models. GUIDE and LID support the ER Model; DDEW is based on an ER Model extended to provide subtyping. SKI and ISIS support models that are functionally very similar to restricted subsets of SDM. SNAP supports a subset of the IFO model.

As indicated by the functionality area of the chart, all of the systems except LID concentrate on schema management, providing capabilities for schema definition, schema browsing, and query formulation. The ISIS and SNAP systems also provide some limited facilities for convenient representation and perusal of printed data. And, in contrast to the other five systems, LID is oriented entirely toward data browsing and allows the user, given a particular object in the database, to traverse schema relationships to find related objects. In LID there is no capability of viewing sets of objects, only individual ones. Therefore, if the user wishes, say, to view all travelers who have been to Japan, the locations visited by each traveler would have to be examined individually. The system will not perform the search and return the result as one conceptual entity.

With respect to the implementations of these six systems, Figure 21 indicates that all but DDEW are experimental research prototypes and DDEW and SKI are the only ones that interface with actual, disk-based databases. DDEW may be used to specify and browse schemas; SKI may also be used to formulate a query on an existing semantic database. Clearly, however, the lack of actual DBMSs underneath these systems does not detract from the significance of their research contributions. Any issues concerning the speed of disk accesses while examining data are largely irrelevant. Finally, we note that these systems indicate a historical trend toward workstation environments.

The final axis in the chart refers to the use of graphics in the six example systems and indicates a progression from fairly simple graph- (or network-) based representations of schemas to richer visual (but still

graphlike) representations. Correspondingly, there has been a development in terms of the naturalness with which these systems allow the user to interact with the schema while browsing and specifying queries. In Foley and Dam [1982] the authors differentiate three sorts of feedback that an interactive system can give: *lexical* (such as the echoing of typed characters), *syntactic* (such as highlighting a selected menu item), and *semantic*. Semantic feedback is the most sophisticated sort of feedback and might, for example, indicate in an obvious fashion that a user-requested operation has been performed (e.g., that a subtype has been added to the schema). In general, the six selected interactive systems provide good semantic feedback, with the visual representation of schemas being dynamically modified as they are manipulated. The trend has been toward richer semantic feedback.

All six systems represent the schema using graph structures. GUIDE and DDEW do this essentially using ER diagrams. In GUIDE the schema is statically structured and is not altered by the user. In DDEW the system suggests an appropriate schema representation, but the user may alter it. In ISIS and SNAP the user directly defines the visual representation of the schema. SKI completely controls the visual structure of the schema for the user. (Of course, *LID*, since it only supports the browsing of individual data items, does not provide visual access to the entire schema.) All these systems (except LID) allow users to pan and zoom, and hide *irrelevant portions* of the schema. SKI, ISIS, and SNAP are now discussed in more detail.

In the SKI system there is no notion of a statically defined graphical layout of the underlying schema. Instead, a graphical representation of the relevant subschema is created dynamically as the user specifies that various connections be displayed. The subschema is placed on a formated screen; the screen is broken into stripes, each containing components with different semantic significance (e.g., one for object types, another for attributes defined on these types). The philosophy behind this is to remove from the user the burden of having to manage a large, complex graph that represents a schema and to automatically format subschemas according to their content. Of course, this approach does not allow the user to return easily to familiar visual representations during a lengthy session.

ISIS and SNAP both take the approach of providing more permanent representations of schemas. In both systems the user creates much of the original visual representation of a schema; this is saved by the system and provides the basis for the display of both the schema and portions of it. Both systems also permit users to modify the representation of schemas.

With respect to the manner in which the user peruses the schema while browsing, all of the systems (except for LID) provide mechanisms for massaging the visual representation directly in order to focus on specific areas of interest within the schema. The user follows semantic connections (e.g., attributes and type/subtype relationships) in order to isolate various sorts of data that relate to each other. This is much more effective than forcing the user to peruse the schema only by navigating up and down, and right and left over a schema that is too large to fit on a screen. Thus, the user may hide levels of detail and/or annotate the schema while browsing. Schneiderman [1980] has coined the phrase *direct manipulation* to characterize this style of interaction, where the user has the feeling of manipulating a real-world object while interfacing with the system. This provides very rich and effective semantic feedback for the user. These systems vary somewhat in their approach to direct manipulation during browsing. ISIS supports operators for maneuvering between its diagram for representing ISA relationships and its diagrams for representing attribute relationships. Also, SKI provides several high-level operators that may be used to make very dramatic shifts in focus, such as requesting to view all areas of the schema that would be affected if a particular data operation was performed.

The six systems also vary in how they support query specification, with a growing tendency toward more graphics-based methods of specifying an entire query.

Figure 21. (Graphlike-based interfaces based on semantic models. Blanks indicate capability not supported to authors' knowledge. Key to references in this figure precedes entries in reference list.

	REFERENCES	DATA MODEL	FUNCTIONALITY: SCHEMA BROWSING	FUNCTIONALITY: SCHEMA DEFINITION	FUNCTIONALITY: QUERY SPECIFICATION	FUNCTIONALITY: DATA BROWSING	IMPLEMENTATION: GRAPHICS INTERFACE	IMPLEMENTATION: INTERFACE TO DBMS	IMPLEMENTATION: SYSTEM EQUIP/OS/IMPL LANG	USE OF GRAPHICS: VISUAL REP OF SCHEMA	USE OF GRAPHICS: PARADIGM DEFINITION	USE OF GRAPHICS: PARADIGM OF SCHEMA VIEWING	USE OF GRAPHICS: QUERY SPECIFICATION
GUIDE	[Wuk82]	ER	✓				PROTOTYPE		VAX/VMS/C	FIXED ER DIAGRAM	NA	DIRECT MANIPULATION OF SCHEMA	DIRECT MANIPULATION OF SCHEMA
LID	[Fog84]	ER					GRAPHICAL	PROTOTYPE	?	ONLY LOCAL STRUCTURE	NA	NA	TRIPLE LEVEL NAVIGATION
DDEW	[DBB84]	ERL		✓			PRODUCTION QUALITY	GENERATES RELATIONAL NETWORK HIERARCHICAL SCHEMAS	JUPITER 12/UNIX/C	ERL DIAGRAMS	TEXT BASED	DIRECT MANIPULATION OF SCHEMA	NA
SKI	[KMB84]	INSYDE SUBSET	✓	✓	✓		PROTOTYPE	SEMBASE DBMS	SUN WORKSTATION/ UNIX/C	FORMATTED TYPE/ SUBTYPE/ ATTRIBUTE DIAGRAMS	GRAPHICAL	SEMANTICALLY DRIVEN OPERATIONS	SPECIFICATION OF DERIVED DATA
ISIS	[GGX85]	SDM SUBSET	✓	✓	✓	TEXT-BASED	PROTOTYPE	IN MEMORY	APOLLO/UNIX/C	TYPE/ SUBTYPE/ ATTRIBUTE DIAGRAMS	GRAPHICAL	MODE ORIENTED	SPECIFICATION OF DERIVED DATA
SNAP	[Bat86]	IFO SUBSET	✓	✓	✓		PROTOTYPE		SYMBOLICS 3600/ ZETA LISP	IFO DIAGRAMS	GRAPHICAL	DIRECT MANIPULATION OF SCHEMA	DIRECT MANIPULATION OF SCHEMA

GUIDE supports selection-like queries; to specify them, the user graphically identifies relevant portions of the schema, and a text-based query is constructed in parallel by the system. SKI was the first system to embody the concept of expressing queries through the iterative specification of derived data, thus making a query essentially an extension of the schema. The user may traverse the schema to find the general type of data of interest and then repeatedly refine this type by specifying subtype predicates. These predicates are defined textually. SKI provides a series of pop-up menus that support an editor that displays the syntax permitted in subtype definitions. ISIS uses a similar paradigm but provides a much more sophisticated editor that allows the user to specify a query in an almost entirely mouse-based fashion. SNAP uses a different approach in which copies of schema components are directly manipulated to form queries. Intuitively, SNAP queries can be viewed as a generalization of Zloof's Query-by-Example [Zloof 1987] to a graphics-based model.

A commercial implementation of a graphics interface based on a semantic model has recently been developed. It is similar in structure and function to SKI, ISIS, and SNAP and includes a schema design tool and a data browser [Rogers and Cattell 1987].

4.4 Theory

Over recent years, there has been increasing interest in the application of theoretical techniques to investigating concepts and problems raised by semantic database models. Considerable work has been performed on constructed types considered in isolation, with a primary focus on data access and manipulation languages for them. Another important topic considers ISA networks in isolation and studies the inference of various properties in them. Other important topics include comparing the data capacity of constructed types and characterizing update propagation in semantic schemas. These topics have been studied using a variety of different models: no one model has emerged as the common basis for theoretical investigations of these issues. In this section we briefly survey the work in these various areas. A more comprehensive survey of work on theoretical research on constructed types may be found in Hull [1987]. Also, we note that some of the theoretical work on the IFO [Abiteboul and Hull 1987] and IRIS [Lyngbaek and Vianu 1987] models is discussed in Section 3.2.

Surprisingly, a significant amount of the research on access languages for constructed types has been performed using an offshoot of the relational model called non-first-normal-form relations [Abiteboul and Bidoit 1986; Fischer and Thomas 1983; Jaeschke and Schek 1982; Makinouchi 1977]. These are relations where some columns may hold relations instead of atomic values; nesting of relations in this way is permitted to arbitrary depths. The structure of such a non-first-normal-form relation may be viewed as a constructed type formed using alternating layers of aggregation and grouping. Calculus-based and algebraic languages for non-first-normal-form relations have been developed [Fischer and Thomas 1983; Jaeschke and Schek 1982] and shown to have equivalent expressive power [Abiteboul and Beeri 1987; Kuper and Vardi 1984].

Another model of constructed types is the Format Model [Hull and Yap 1984]. Formats are built hierarchically from three constructs: aggregation, grouping, and marked or disjoint union. The third construct is used to model types that result from generalizations involving two or more disparate types. Original work on the Format Model focused on comparing the data capacity of formats [Hull and Yap 1984; O'Dunlaing and Yap 1982]. A more recent paper [Abiteboul and Hull 1986] proposes a query language for formats that is based on a rewrite operator.

The *Logical Data Model* (LDM) [Kuper and Vardi 1984, 1985; Kuper 1985] can be viewed as a further extension of constructed types. LDM was introduced primarily to provide a logic-based model lying between the record-oriented models (relational, network, hierarchical) and actual physical implementation, but issues studied

with the LDM are also important from the point of view of semantic models.

As with the Format Model, LDM schemas are built from basic types and type constructors for aggregation, grouping, and marked union. (In Kuper and Vardi [1984, 1985] marked union is not included.) Unlike the constructed types of GSM and the Format Model, however, schemas of the LDM are directed graphs rather than trees. Instances of LDM schemas are defined using the complementary notions of *l-* and *r-values* from the theory underlying programming language assignment statements. This formalism captures the object-oriented nature of semantic models in a novel manner: *l*-values correspond to object identities, while *r*-values correspond to their values. In Kuper and Vardi [1984, 1985] calculus-based and algebraic manipulation languages are introduced for LDM and shown to have equivalent expressive power. In Kuper and Vardi [1985], the relative data capacity of LDM schemas is studied.

We now turn to research focused primarily on ISA relationships. Two recent papers study the interplay of ISA relationships in connection with integrity constraints [Atzeni and Parker 1986; Lenzerini 1987]. Both of these works use a simple abstract semantic model based on abstract types and ISA relationships. In Atzeni and Parker [1986] the interaction of ISA relationships and disjointness constraints is studied, and a sound and complete set of inference rules for these properties is presented. Also, it is shown that various problems are decidable in polynomial time. One such problem is the *satisfiability* problem, which concerns whether a node in a schema is nonempty in at least one instance. In Lenzerini [1987], covering constraints are studied in addition to disjointness constraints. In this case many problems are NP-complete, including the satisfiability problem.

We now turn to the problem of determining how atomic updates propagate in semantic schemas. This is of particular interest because it considers the various constructs of semantic models taken together rather than in isolation. As a simple example of update propagation, suppose that in the World Traveler Database Pam is both a person and a tourist. If Pam is deleted from PERSON, she should also be deleted from TOURIST. Update propagation relative to the basic structural components of semantic models has been studied in the context of both FDM [Hechenberg and Kerschberg 1981] and IFO [Abiteboul and Hull 1985, 1987]. In both papers the semantics of update propagation is broken into two logical pieces: one concerned with the impact of updates on the local constructs of a schema and the other with the global impact implied by their combination. The overall impact of an update at a given node is essentially defined to be the sum of the impacts implied by the local update semantics. In both models, acyclicity conditions on ISA relationships ensure that each node need be visited at most once during this computation.

5. CONCLUDING REMARKS

In this paper we have surveyed a wide area of research, all of it centered around semantic database modeling. We have taken an in-depth look at the fundamental motivations and aspects of semantic models and examined a number of specific models. Further, several research directions that are based on semantic models have been discussed, including semantic data access languages, graphical database interfaces based on semantic models, physical implementations of semantic DBMSs, and theoretical investigations of semantic models.

Clearly, there are many more research issues relating to semantic models that could be investigated, such as the integration of temporal reasoning into semantic models, the optimization of semantic database queries, the development of semantic database machines, and the construction of expert database systems that use semantic models (such databases would be capable of making inferences about complex semantic data). However, although some research is currently being conducted in these areas, it has not reached the level of maturity appropriate for a survey paper.

We would, however, like to conclude this paper by mentioning a rapidly growing area of database research that is related to semantic modeling. Recently, a number of

research projects have focused on the development of data models that are more expressive than conventional models but use techniques different from those of semantic models. Experimental systems based on these *object-oriented* models are typically centered around the concepts of large objects and extensible type structures, such as arise in engineering design applications, and the concept of local behavior stemming from object-oriented programming languages.

Object-oriented database models are fundamentally different from semantic models in that they support forms of local behavior in a manner similar to object-oriented programming languages. This means that a database entity may locally encapsulate a complex procedure or function for specifying the calculation of a data operation. This gives the database user the capability of expressing, in an elegant fashion, a wider class of derived information than can be expressed in semantic models. For example, in the system Postgres [Stonebraker and Rowe 1986] a data item may have an attribute that is a database query or an application program. This project is an experiment in extending already-developed relational techniques to the handling of complex data. In the system Cactis [Hudson and King 1986], an object may have attribute values that are computed by arbitrary computable functions. This project focuses on the design of formalisms that may be used to implement complex derived data efficiently and uses attributed graphs as an underlying physical construct. The Gemstone [Maier et al. 1986] project focuses on providing database users with message-passing mechanisms similar to those of Smalltalk. Thus, generalized methods may be defined for specifying how an object should react to messages from another object.

On another dimension, object-oriented models are similar to semantic models in that they provide mechanisms for constructing complex data by interrelating objects. The Exodus [Carey et al. 1986] system attempts to support the storage and access of very large objects. The Exodus project also provides the capability of allowing the user to easily extend the type structure. Another extensible system, designed for such applications as engineering, is Probe [Manola and Dayal 1986]. The GENESIS project [Batory et al. 1988] focuses on the rapid development of customized database management systems.

Speaking broadly, semantic modeling has concentrated largely on building complex data via mechanisms like attributes, aggregation, and generalization, which are widely viewed to be adequate for most business and commercial applications. In contrast, object-oriented models are oriented toward novel applications that must support complex domains such as software design [Hudson and King 1987], VLSI and printed circuit board design, and CAD/CAM [Andrews and Harris 1987; Su et al. 1988]. These applications are generally interactive and require highly dynamic database systems where the user may control local behavior and dynamically modify the type structure. Software specifications, text, and engineering designs are also much larger objects than typical business objects. Other novel research issues also arise in the context of object-oriented databases, including very long transactions (to support interactive design), nested transactions (to support complex design functions), and mechanisms for obtaining multiple blocks of data from a mass storage device quickly (to allow the efficient retrieval of large objects). Research on these and related topics will be crucial in expanding the usefulness of database systems to nontraditional, nonbusiness applications.

ACKNOWLEDGMENTS

We would like to acknowledge the anonymous referees and Salvatore March for their very helpful comments concerning previous versions of this survey. Richard Hull was supported in part by the National Science Foundation under grants IST-83-06517 and IST-85-11541. Roger King was supported in part by the Office of Naval Research under contract number N00014-86-K-0054 and by the National Science Foundation under grant DMC-8505164.

REFERENCES

[AbB87] ABITEBOUL, S., AND BEERI, C. 1987. On the power of languages for the manipulation of complex objects (extended abstract). In *Proceedings of International Workshop on Nested Relations and Complex Objects*. (Darmstadt, Germany, Apr.). INRIA, Roquencourt, France.

[AbB84] ABITEBOUL, S., AND BIDOIT, N. 1986. Nonfirst normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci. 33*, 361–393.

[AbH85] ABITEBOUL, S., AND HULL, R. 1985. Update propagation in the IFO database model. In *Proceedings of the International Conference on Foundations of Data Organization*. Organizing Committee of the International Conference on Foundations of Data Organization, c/o S. P. Ghosh, IBM Research, Almaden, Calif., pp. 243–251.

[AbH86] ABITEBOUL, S. AND HULL, R. 1986. Restructuring of complex database objects and office forms. In *Proceedings of the International Conference on Database Theory* (Rome, Sept.).

[AbH87] ABITEBOUL, S., AND HULL, R. 1987. IFO: A formal semantic database model. *ACM Trans. Database Syst. 12*, 4 (Dec.), 525–565.

[Abr74] ABRIAL, J. R. 1974. Data semantics. *Data Base Management*. North-Holland, Amsterdam, pp. 1–59.

[AfM84] AFSARMANESH, H., AND MCLEOD, D. 1984. A framework for semantic database models. In *Proceedings of the NYU Symposium on New Directions for Database Systems* (New York, May 16–18). New York Univ., New York.

[Aik85] AIKENS, J. 1985. A representation scheme using both frames and rules. In *Rule-Based Expert Systems*, B. Buchanan and E. Shortliffe, Eds. Addison-Wesley, Reading, Mass., pp. 424–440.

[ACO85] ALBANO, A., CARDELLI, L., AND ORSINI, R. 1985. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. Database Syst. 10*, 2 (June), 230–260.

[AOO85] ALBANO, A., OCCHIUTO, M. E., AND ORSINI, R. 1985. Galileo Reference Manual, VAX/UNIX Version 1.0. Tech. Rep. Dipartimento di Informatica, Univ. di Pisa, Pisa, Italy.

[AnH87] ANDREWS, T., AND HARRIS, C. 1987. Combining language and database advances in an object-oriented development environment. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Oct.), pp. 430–440.

[AtB87] ATKINSON, M. P., AND BUNEMAN, O. P. 1987. Database programming languages. *ACM Comput. Surv. 19*, 2 (June), 105–190.

[AtK83] ATKINSON, M. P., AND KULKARNI, K. G. 1983. Experimenting with the functional data model. Tech. Rep. Persistent Programming Research Rep. 5, Univ. of Edinburgh, Edinburgh, Scotland.

[AtP86] ATZENI, P., AND PARKER, D. S. 1986. Formal properties of net-based knowledge representation schemes. In *Proceedings of the 2nd IEEE International Conference on Data Engineering*. IEEE, New York, pp. 700–706.

[Bac78] BACKUS, J. 1978. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug.), 613–641.

[BLN86] BATINI, C., LENZERINI, M., AND NAVATHE, S. B. 1986. A comparative analysis of methodologies for Database schema integration. *ACM Comput. Surv. 18*, 4 (Dec.), 323–364.

[BaK85] BATORY, D. S., AND KIM, W. 1985. Modeling concepts for VLSI CAD objects. *ACM Trans. Database Syst. 10*, 3 (Sept.), 322–346.

[BBG] BATORY, D. S., BARNETT, J. R., GARZA, F. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. 1988. GENESIS: A reconfigurable database management system. *IEEE Trans. Softw. Eng.* to appear.

[BKK86] BOBROW, D., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., AND ZDYBEL, F. 1986. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Mar.). ACM, New York, pp. 17–29.

[Bor85] BORGIDA, A. 1985. Features of languages for the development of information systems at the conceptual level. *IEEE Software 2*, 1 (Jan.), 63–72.

[BPP76] BRACCHI, G., PAOLINI, P., AND PELAGATTI, G. 1976. Binary logical associations in data modelling. In *Modelling in Data Base Management Systems*. North Holland, Amsterdam, pp. 125–148.

[BrS85] BRACHMAN, R. J., AND SCHMOLZE, J. G. 1985. An overview of the KL-ONE knowledge representation system. *Cognitive Sci. 9* (1985), 171–216.

[Bro84] BRODIE, M. L. 1984. On the development of data models. In *On Conceptual Modelling*, M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Eds. Springer-Verlag, New York, pp. 19–48.

[BrR84] BRODIE, M. L., AND RIDJANOVIC, D. 1984. On the design and specification of database transactions. In *On Conceptual Modelling*. Springer-Verlag, New York, pp. 277–306.

[BMS84] BRODIE, M. L., MYLOPOULOS, J., AND SCHMIDT, J. W. Eds. 1984. *On Conceptual Modelling*. Springer-Verlag, New York.

[BrP83] BROWN, R., AND PARKER, D. S. 1983. LAURA: A formal data model and her logical design methodology. In *Proceedings of the 9th International Conference on Very Large Data Bases*. Very Large Database Endowment, Saratoga, Calif., pp. 206–218.

[BFN82] BUNEMAN, P., FRANKEL, R. E., AND NIKHIL, R. 1982. An implementation technique for database query languages. *ACM Trans. Database Syst. 7*, 2 (June), 164–186.

[CDR86] CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J. 1986. Object and file

management in the EXODUS extensible database system. In *Proceedings of the 12th International Conference on Very Large Databases* (Aug.). Very Large Database Endowment, Saratoga, Calif., pp. 91–100.

[CGT75] CHAMBERLIN, D. D., GRAY, J. N., AND TRAIGER, I. L. 1975. Views, authorization and locking in a relational database system. In *Proceedings of AFIPS National Computer Conference*, vol. 44. AFIPS Press, Reston, VA., pp. 425–430.

[CDF83] CHAN, A., DAYAL, U., FOX, S., AND RIES, D. 1983. Supporting a semantic data model in a distributed database system. In *Proceedings of the 9th International Conference on Very Large Data Bases*. Very Large Database Endowment, Saratoga, Calif., pp. 354–363.

[CDF82] CHAN, A., DANBERG, S., FOX, S. LIN, W. T. K., NORI, A., AND RIES, D. 1982. Storage and access structures to support a semantic data model. In *Proceedings of the 8th International Conference on Very Large Data Bases*. Very Large Database Endowment, Saratoga, Calif., pp. 122–130.

[Che76] CHEN, P. P. 1976. The entity-relationship model–Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar.), 9–36.

[Cod79] CODD, E. F. 1979. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec.), 397–434.

[CoP84] COSMADAKIS, S. S., AND PAPADIMITRIOU, C. H. 1984. Updates of relational views. *J. ACM 31*, 4 (Oct.), 743–760.

[DDG86] DARDAILLER, P., DELOBEL, C., AND GRAUDIN, J. P. 1985. Modelisation progressive d'une base de donnees. Tech. Rep. 493, Laboratoire de Genie Informatique.

[Dat81] DATE, C. J. 1981. *An Introduction to Database Systems*, vol 1. Addison-Wesley, Reading, Mass.

[DaH84] DAYAL, U., AND HWANG, H. Y. 1984. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. Softw. Eng. SE-10*, 6, 628–644.

[DHP74] DEHENEFFE, C., HENNEBERT, H., AND PAULUS, W. 1974. Relational model for a data base. In *Proceedings of the IFIP Congress*. pp. 1022–1025.

[DKL85] DERRETT, N., KENT, W., AND LYNGBAEK, P. 1985. Some aspects of operations in an object-oriented database. *IEEE Database Eng. Bull. 8*, 4 (Dec.).

[Fag77] FAGIN, R. 1977. Multivalued dependencies and a new normal form for relation databases. *ACM Trans. Database Syst. 2*, 3 (Sept.), 262–278.

[FKM85] FARMER, D. B., KING, R., AND MYERS, D. A. 1985. The semantic database constructor. *IEEE Tran Softw. Eng. SE-11*, 7, 583–591.

[FiK86] FIKES, R., AND KEHLER, T. 1985. The role of frame-based representation in reasoning. *Commun. ACM 28*, 9 (Sept.), 904–920.

[Fin79] FINDLER, N., Ed. 1979. *Associative Networks*. Academic Press, New York.

[FIT83] FISCHER, P., AND THOMAS, S. 1982. Operators for non-first-normal-form relations. In *Proceedings of the 7th International Computer Software Applications Conference* (Chicago, Nov.). IEEE, New York, pp. 464–475.

[Fog84] FOGG, D. 1984. Lesson from a "Living in a database" graphical query interface. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Boston, Mass.). ACM, New York, pp. 100–106.

[FoD82] FOLEY, J. D., AND VAN DAM, A. 1982. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass.

[GoR83] GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass.

[GGK85] GOLDMAN, K. J., GOLDMAN, S. A., KANELLAKIS, P. C., AND ZDONIK, S. B. 1985. ISIS: Interface for a semantic information system. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York, pp. 328–342.

[Gut77] GUTTAG, J. 1977. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (June), 396–404.

[HaL74] HAINAUT, J. L., AND LECHARLIER, B. 1974. An extensible semantic model of database and its data language. In *Proceedings of the IFIP Congress*, pp. 1026–1030.

[HaB80] HAMMER, M., AND BERKOWITZ, B. 1980. DIAL: A programming language for data intensive applications. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York, pp. 75–92.

[HaM81] HAMMER, M., AND MCLEOD, D. 1981. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept.), 351–386.

[HeK81] HECHT, M. S., AND KERSCHBERG, L. 1981. Update semantics for the functional data model. Tech. Rep., Bell Laboratories, Holmdel, N.J.

[HuK86] HUDSON, S. E., AND KING, R. 1986. CACTIS: A database system for specifying functionally-defined data. In *Proceedings of the Workshop on Object-Oriented Databases* (Asilomar, Pacific Grove, Calif., Sept.). IEEE, New York.

[HuK87] HUDSON, S. E., AND KING, R. 1987. Object-oriented database support for software environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, Calif., May). ACM, New York, pp. 491–503.

[Hul87] HULL, R. 1987. A survey of theoretical research on typed complex database objects. In *Databases*, J. Paredaens, Ed. Academic Press, London.

[HuY84] HULL, R., AND YAP, C. K. 1984. The format model: A theory of database organization. *J. ACM 31*, 3 (July), 518–537.

[IsB84] ISRAEL, D. J., AND BRACHMAN, R. J. 1984. Some remarks on the semantics of representation languages. In *On Conceptual Modelling*. Springer-Verlag, New York, pp. 119–146.

[JaS82] JAESCHKE, B., AND SCHEK, H. J. 1982. Remarks on the algebra of non first normal form relations. In *Proceedings of the ACM Symposium on Principles of Database Systems*. ACM, New York.

[KeC83] KEHLER, T. P., AND CLEMENSON, G. D. 1983. An application development system for expert systems. *Syst. Softw. 3*, 1 (Jan.), 212–223.

[Ken78] KENT, W. 1978. *Data and Reality*. North-Holland, Amsterdam.

[Ken79] KENT, W. 1979. Limitations of record-based information models. *ACM Trans. Database Syst. 4*, 1 (Mar.), 107–131.

[KoP76] KERSCHBERG, L., AND PACHECO, J. E. S. 1976. A functional data base model. Tech. Rep., Pontificia Univ. Catolica do Rio de Janeiro, Rio de Janeiro, Brazil.

[KKT76] KERSCHBERG, L., KLUG, A., AND TSICHRITZIS, D. 1976. A taxonomy of data models. In *Systems for Large Data Bases*. North-Holland, Amsterdam, pp. 43–64.

[KhC86] KHOSHAFIAN, S. N., AND COPELAND, G. P. 1986. Object identity. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications* (Portland, Oreg., Sept.). ACM, New York, pp. 406–416.

[Kin84] KING, R. 1984. Sembase: A semantic DBMS. In *Proceedings of the First International Workshop on Expert Database Systems* (Oct.). Univ. of Southern Carolina, Columbia, S.C., pp. 151–171.

[KiM82] KING, R., AND MCLEOD, D. 1982. The event database specification model. In *Proceedings of the 2nd International Conference on Databases: Improving Usability and Responsiveness* (Jerusalem, Israel), pp. 299–321.

[KiM84] KING, R., AND MCLEOD, D. 1984. A unified model and methodology for conceptual database design. In *On Conceptual Modelling*. Springer-Verlag, New York, pp. 313–331.

[KiM85a] KING, R., AND MCLEOD, D. 1985a. A database design methodology and tool for information systems. *ACM Trans. Off. Inf. Syst. 3*, 1 (Jan.), 2–21.

[KiM85b] KING, R., AND MCLEOD, D. 1985b. Semantic database models. In *Database Design*. Springer-Verlag, New York, pp. 115–150.

[Kup85] KUPER, G. M. 1985. The logical data model: A new approach to database logic. Ph.D. dissertation, Computer Science Dept., Stanford Univ.

[KuV84] KUPER, G. M., AND VARDI, M. Y. 1984. The logical data model. In *Proceedings of ACM SIGACT News–SIGMOD Symposium on Principles of Database Systems*. ACM, New York, pp. 86–96.

[KuV85] KUPER, G. M., AND VARDI, M. Y. 1985. On the expressive power of the logical data model (extended abstract). In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York.

[LaR82] LANDERS, T. A., AND ROSENBERG, R. L. 1982. An overview of multibase. *Distributed Databases*. North-Holland, Amsterdam.

[Len87] LENZERINI, M. 1987. Covering and disjointness constraints in type networks. In *Proceedings of the IEEE Conference on Data Engineering* (Los Angeles, Calif.). IEEE, New York, pp. 386–393.

[LSA77] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. 1977. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug.), 564–576.

[LyK86] LYNGBAEK, P., AND KENT, W. 1986. A data modeling methodology for the design and implementation of information In *International Workshop on Object-Oriented Database Systems* (Asilomar, Pacific Grove, Calif., Sept.). IEEE, New York.

[LyV87] LYNGBAEK, P., AND VIANU, V. 1987. Mapping a semantic database model to the relational model. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Francisco, Calif., June). ACM, New York.

[Mac85] MACGREGOR, R. M. 1985. ARIEL—A semantic front-end to relational DBMSs In *Proceedings of the 11th International Conference on Very Large Data Bases*. Very Large Data Base Foundation, Saratoga, Calif., pp. 305–315.

[MSO86] MAIER, D., STEIN, J., OTIS, A., AND PURDY, A. 1986. Development of an object-oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Sept. 29–Oct. 2). ACM, New York, pp. 472–482.

[Mak77] MAKINOUCHI, A. 1977. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of the 3rd International Conference on Very Large Databases* (Tokyo, Oct.), pp. 447–453.

[MaD86] MANOLA, F., AND DAYAL, U. 1986. PDM: An object-oriented data model. In *Proceedings of the Workshop on Object-Oriented Databases* (Pacific Grove, Calif., Sept. 23–26). IEEE, New York, pp. 18–25.

[MaP86] MARYANSKI, F., AND PECKHAM, J. 1986. Semantic data models. Tech. Rep. CSTR 86-15, Dept. of Computer Science and Engineering, Univ. of Connecticut, Storrs, Conn.

[Min84] MINSKY, M. L. 1984. A framework for representing knowledge. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York, pp. 211–277.

[Moo86] MOON, D. A. 1986. Object-oriented programming with flavors. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York, pp. 1–8.

[Myl80] MYLOPOULOS, J. 1980. An overview of knowledge representation. In *Workshop on Data Abstract, Databases, and Conceptual Modelling* (Pingree Park, Colo.). ACM, New York, pp. 5–12.

[MBW80] MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. 1980. A language facility for designing database-intensive applications. *ACM Trans Database Syst. 5*, 2 (June), 185–207.

[MBG86] MYLOPOULOS, J., BORGIDA, A., GREENSPAN, S., MEGHINI, C., AND NIXON, B. 1986. Knowledge representation in the software development process: A case study. Tech. Rep., Univ. of Toronto, Toronto, Canada.

[NEL86] NAVATHE, S., ELMASRI, R., AND LARSON, J. 1986. Integrating user views in database design. *IEEE Computer 19*, 1, 50–62.

[Nik84] NIKHIL, R. 1984. An incremental, strongly typed applicative programming system for databases. Ph.D. dissertation. Dept. of Computer and Information Sciences, Univ. of Pennsylvania, Philadelphia.

[O'Y82] O'DUNLAING, C., AND YAP, C. K. 1982. Generic transformation of data structures. in *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 186–195.

[Ris85] RISHE, N. 1985. Semantic modeling of data using binary schemata. Tech. Rep. TRCS85-06. Univ. of California, Santa Barbara, Calif.

[Ris86] RISHE, N. 1986. On representation of medical knowledge by a binary data model. In *Proceedings of the 5th International Conference on Mathematical Modelling*, X. J. R. Avula, G. Leitman, C. D. Mote, Jr., and E. Y. Rodin, Eds. Pergamon Press, Elmsford, N.Y.

[RoC87] ROGERS, T. R., AND CATTELL, R. G. G. 1987. Entity-relationship database user interfaces. Tech. Rep. Sun Microsystems, Mountain View, Calif.

[RoY84] ROUSSOPOULOS, N., AND YEH, R. T. 1984. An adaptable methodology for database design. *IEEE Computer* (May), 64–80.

[RoS79] ROWE, L., AND SCHOENS, K. A. 1979. Data abstractions, views, and updates in RIGEL. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York, pp. 71–81.

[Sch80] SCHNEIDERMAN, B. 1980. *Software Psychology: Human Factors in Computer and Information Systems*. Prentice-Hall, Englewood Cliffs, N.J.

[Sen75] SENKO, M. E. 1975. Information systems: Records, relations, set, entities, and things. *Inf. Syst. 1*, 1, 3–13.

[Shi81] SHIPMAN, D. 1981. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar.), 140–173.

[Sho82] SHOSHANI, A. 1982. Statistical databases: Characteristics, problems, and some solutions. In *Proceedings of the 8th International Conference on Very Large Data Bases* (Mexico City). Very Large Data Base Endowment, Saratoga, Calif., pp. 208–222.

[SiK77] SIBLEY, E. H., AND KERSCHBERG, L. 1977. Data architecture and data model considerations. In *Proceedings of the National Computer Conference*. AFIPS Press, Reston, Va., pp. 85–96.

[SmS77] SMITH, J. M., AND SMITH, D. C. P. 1977. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (June), 105–133.

[SFL81] SMITH, J. M., FOX, S., AND LANDERS, T. 1981. Reference manual for ADAPLEX. Tech. Rep., Computer Corporation of America.

[SBD81] SMITH, J. M., BERNSTEIN, P. A., DAYAL, U., GOODMAN, N., LANDERS, T., LIN, K. W. T., AND WONG, E. 1981. Multibase–Integrating heterogeneous distributed database systems. In *Proceedings of AFIPS National Computer Conference*. AFIPS Press, Reston, Va., pp. 487–499.

[SBM83] STEFIK, M., BOBROW, D. G., MITTAL, S., AND CONWAY, L. 1983. Knowledge programming in LOOPS: Report on an experimental course. *Artif. Intell. 4*, 3, 3–14.

[StR86] STONEBRAKER, M., AND ROWE, L. A. 1986. The design of postgres. In *Proceedings of International Conference on the Management of Data* (May). ACM, New York, pp. 340–355.

[SWK76] STONEBRAKER, M. R., WONG, E., KREPS, P., AND HELD, G. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept.), 189–222.

[Su83] SU, S. Y. W. 1983. SAM*: A semantic association model for corporate and scientific statistical databases. *Inf. Sci. 29*, 151–199.

[Su86] SU, S. Y. W. 1986. Modeling integrated manufacturing data with SAM*. *IEEE Computer Magazine* (Jan.), 34–49.

[Su80] SU, S. Y. W., AND LO, D. H. 1980. A semantic association model for conceptual database design. In *Entity-Relationship Approach to Systems Analysis and Design*. North Holland, Amsterdam, pp. 147–171.

[SK88] SU, S. Y. W., KRISHNAMURTHY, V., AND LAM, H. 1988. An object-oriented semantic association model (OSAM*). In *AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications*, S. Kumara and R. L. Kashyap, Eds. American Institute of Industrial Engineers, Norcross, Ga.

[TYF86] TEOREY, T. J., YANG, D., AND FRY, J. P. 1986. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput Surv. 18*, 2 (June), 197–222.

[TsK77] TSICHRITZIS, D., AND KLUG, A. C. 1977. *American National Standards Institute/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems*. AFIPS Press, Reston, Va.

[TsL82] TSICHRITZIS, D. C., AND LOCHOVSKY, F. H. 1982. *Data Models*. Prentice-Hall, Englewood Cliffs, N.J.

[TsZ84] TSUR, S., AND ZANIOLO, C. 1984. An implementation of GEM–Supporting a semantic data model on a relational back-end. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York, pp. 286–295.

[Ull82] ULLMAN, J. D. 1982. *Principles of Database Systems*. 2nd ed. Computer Science Press, Rockville, MD.

[Ull87] ULLMAN, J. 1987. Database theory: Past and future. In *Proceedings of ACM SIGACT News–SIGMOD-SIGART Principles of Database Systems*. (San Diego, Calif., Mar.). ACM, New York.

[UrD86] URBAN, S. D., AND DELCAMBRE, L. M. L. 1986. An analysis of the structural, dynamic, and temporal aspects of semantic data models. In *Proceedings of the 2nd IEEE International Conference on Data Engineering* (Feb.). IEEE, New York, pp. 382–389.

[VeB82] VERHEIJEN, G., AND BEKKUM, J. V. 1982. NIAM: An information analysis method. In *Information Systems Design Methodologies: A Comparative Review*, T. Olle, H. Sol and A. Verrijn-Stuart, Eds. IFIP.

[Via87] VIANU, V. 1987. Dynamic functional dependencies and database aging. *J. ACM 34*, 1 (Jan.), 28–59.

[Was79] WASSERMAN, A. I. 1979. The data management facilities of PLAIN. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York.

[WiE80] WIEDERHOLD, G., AND EL-MASRI, R. 1980. Structural model for database design. In *Entity-Relationship Approach to Systems Analysis and Design*. North Holland, Amsterdam.

[WoK82] WONG, H. K. T. AND KOU, I. 1982. GUIDE: A graphical user interface for database exploration. In *Proceedings of 8th International Conference on Very Large Data Bases*. Very Large Data Base Endowment, Saratoga, Calif., pp. 22–32.

ZANIOLO, C. 1976. Analysis and design of relational schemata for database systems. Tech. Rep. UCLA-Eng-7668, Dept. of Computer Science, Univ. of California at Los Angeles.

ZANIOLO, C. 1983. The database language GEM. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM, New York, pp. 207–217.

[Zlo77] ZLOOF, M. 1977. Query-by-example: A data base language. *IBM Syst. J. 16*, 324–343.

Received May 1986; final revision accepted December 1987.

About the Authors . . .

Richard Hull received his Ph.D. in mathematics from the University of California at Berkeley in 1979. He subsequently joined the Computer Science Department at the University of Southern California and is currently an associate professor there.

Professor Hull's research focuses primarily on theoretical aspects of databases, including complex database objects, semantic database models, deductive databases, and object-oriented database models.

Roger King received his Ph.D. in computer science from the University of Southern California in 1982. Since then he has been an assistant professor in the Computer Science Department at the University of Colorado in Boulder.

Professor King's research efforts are directed largely toward database support for software environments, semantic and object-oriented database implementation, graphical interfaces to databases, the support of derived data in databases, distributed database implementation, the design of rule-based databases, forms-based office interfaces, and computers and law. His research has been supported by the National Science Foundation, the Office of Naval Research, the Naval Ocean Systems Center, the Department of Energy, IBM, and USWest.

Daniel S. Hirschberg received the B.E. degree in electrical engineering from the City College of New York, New York, in 1971, the M.S.E. and M.A. degrees from Princeton University, Princeton, New Jersey, in 1973, and the Ph.D. degree from Princeton University in 1975. He currently is professor and associate chair of graduate studies in the Department of Information and Computer Science at the University of California, Irvine. His present research interests are the design and analysis of combinatorial algorithms, and data structures. Dr. Hirschberg is a member of ACM and SIGACT.

Debra A. Lelewer received the B.S. degree in mathematics from Michigan State University, East Lansing, Michigan, in 1973 and M.S. degrees in mathematics and computer science from California State Polytechnic University, Pomona, California, in 1976 and 1985, respectively. She is currently pursuing a Ph.D. degree at the University of California, Irvine, and is assistant professor in the Computer Science Department at California State Polytechnic University, Pomona. Her present research interests are the design and analysis of algorithms, and data structures. Ms. Lelewer is a member of ACM, SIGACT, and SIGCSE.

The Cactis Project: Database Support for Software Environments

SCOTT E. HUDSON, MEMBER, IEEE, AND ROGER KING, MEMBER, IEEE

Abstract—**The Cactis project is an on-going effort oriented toward extending database support from traditional business oriented applications to software environments. The main goals of the project are to construct an appropriate data model, and develop new techniques to support the unusual data management needs of software environments, including program compilations, software configurations, load modules, project schedules, software versions, nested and long transactions, and program transformations. The ability to manage derived information is a common theme running through many of these unusual data needs, and the Cactis database management system is unique in its ability to represent and maintain derived data in a time and space efficient fashion. A central contribution of Cactis is its integration of the type constructors of semantic models and the localized behavior capabilities of object-oriented database management systems. The Cactis database management system is nearing completion.**

Index Terms—**Database management system, software environments.**

I. INTRODUCTION

THE goal of the Cactis project is to extend the usefulness of database technology from traditional database applications to engineering, and in particular, software and software environments [5], [9], [39], [41], [45]. This means developing database facilities for the management of the myriad of unusual processes and data types involved in the support of the software lifecycle, both for small scale programming tasks, and for programming in the large [11]. These include the design, coding, and debugging of computer programs, as well as the creation, maintenance, and reuse of modules and versions. Software environments are also designed to manage documentations, requirements specifications, schedules, bug reports, test data, etc.

Researchers have noted the unique database requirements of software environments [2], [21]. In this paper, we describe the general scope of the Cactis project, which involves the development of a DBMS called Cactis, as well as the construction of Cactis application software designed to support software environments. We discuss the manner in which the Cactis DBMS provides a unified approach to satisfying many of the needs of software engineering. Cactis, as it stands, is a multiuser centralized database management system. A brief preliminary report on Cactis appears in [20].

In Section II of this paper, we discuss the Cactis DBMS. Our focus is not on the construction of database systems, but on the special data modeling capabilities of Cactis and how they can be used to support the software lifecycle. Therefore, in Section II we concentrate on the Cactis data model and the methods Cactis uses to manage complex data. The data model of Cactis includes powerful type constructors necessary to model such objects as programs and program versions. Unlike business applications, there are also many forms of derived data in a software environment. These can include coarse grained data such as program compilations, software configurations, and load modules, as well as fine grained data such as the control and data flow properties of individual modules, statements, or expressions. The type constructors found in Cactis are derived from semantic databases [24], [29]. To manage derived information, Cactis uses techniques in the spirit of object-oriented databases [13], whereby the rules necessary to calculate derived values are embedded in database objects. Thus, Cactis is both a semantic and an object-oriented database.

In the third section, we describe on-going efforts to apply Cactis to software environment support. This work is in progress and is incomplete. Our goal is to construct a real-life software application on top of Cactis, and to use it as a way of evolving the functionality and implementation of Cactis. In Section III we focus on how Cactis may be used to support the unusual forms of data manipulation required by a software environment DBMS. The needs of a software environment database range over a broad spectrum of capabilities. For example, a software environment must support fine grained data about individual modules and statements for use in optimization of code within a compiler; a database can simplify this task by allowing a program to be viewed as a number of data objects, and by directly supporting the primitives needed to implement data flow analysis.

To support aspects of programming in the large within the same framework, the database should also support the manipulation of programs as a whole and even entities larger than single programs. For example, a common software tool is the "make" capability found in UNIX®

Manuscript received January 15, 1988. This work was supported by the Office of Naval Research under Contract N00014-86-K-0054 and by the National Science Foundation under Grant DMC-8505164.

S. E. Hudson is with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

R. King is with the Department of Computer Science, University of Colorado, Boulder, CO 80309.

IEEE Log Number 8820968.

®UNIX is a registered trademark of AT&T Bell Laboratories.

0098-5589/88/0600-0709$01.00 © 1988 IEEE

[15] and other environments [8], which is used to control recompilation of programs based on last modification times and mutual dependencies. Cactis can easily be programmed to perform these tasks. Similarly, Cactis can be used to manage entities which represent schedules and other management data which may transcend a single program. Another example of special requirements placed on a software environment DBMS is that the user is likely to desire database transactions whose durations are much longer than traditional business transactions. A typical transaction might be a program bug fix, which could involve a long, interactive period of updating a program, then the processes of recompiling the program and reconfiguring any system which uses it. This may also call for the use of nested transactions, to support the many subactivities involved in this procedure.

To date, the main contributions of the Cactis project are the integration of semantic and object-oriented database mechanisms, and the ability to manage derived data in a space and time efficient fashion. The mechanisms used within a Cactis database to manage derived data are based loosely on attribute grammar techniques used in compiler construction [32], [33], as well as from more recent work on incremental attribute evaluation [12], [43] used in syntax directed editors. These mechanisms may be used to implement, in a uniform fashion, the various forms of derived data found in a software environment. Another contribution of Cactis is an efficient rollback and recovery mechanism, which is of primary importance in order to support long nested transactions and versions.

The fourth section of this paper describes the current status of Cactis. It also describes future goals, and in particular discusses the manner in which we intend to experiment with our prototype system. As little is known about the sorts of real-life data that will be found in environments of the future, we propose instrumenting Cactis with software which may be used to gather statistical information. We will then distribute Cactis for experimentation. In this way, we may learn critical information concerning such things as data object sizes, transaction types and durations, and rollback and recovery needs. This will allow us to evolve Cactis into a more useful system.

In sum, the Cactis project clearly has very broad research goals. It is intended to support the database needs of the entire software lifecycle, and thus incorporates much of the functionality of a software environment. In this paper, we address the unique data needs of environments, and show how the Cactis DBMS is able to support them with a small set of data modeling and data manipulation mechanisms. The general goal of the project is to centralize all of the database functionality of a software environment. This will minimize the effort needed to construct environments, and will help allow the design, use, reuse, and maintenance of software to be performed efficiently.

II. The Cactis DBMS

The term "object-oriented" is a widely used term, with many interpretations. It is also a very popular term at this time. While we do not claim that Cactis meets everyone's definition of object-oriented, we do feel that it reflects the two major interpretations of the term. It supports static (or structural) objects, by providing type constructors along the lines of a semantic model. Cactis also allows objects to have local behavior, in the form of procedurally-derived attributes. Below, we describe the Cactis data model and the manner in which is has been implemented in order to ensure a reasonable level of efficiency. As the goal of this paper is to describe the application of Cactis, and not the details of its implementation, we describe the system only enough to provide the proper background for Section III.

Other researchers have built object-oriented database systems. A number of object-oriented database research projects are described in [13]. These projects range from database implementations of the message passing paradigm of Smalltalk [35], [36] to extensible systems which are designed to support objects of varying size [6], to systems designed to support general engineering and multimedia applications [37], [50]. We feel that Cactis is unique in its clean intermixing of structural and behavioral mechanisms, and in its efficient implementation.

A. The Cactis Data Model

Traditionally, database applications have used database systems which are based on simple record-oriented models. Models, such as the network or relational models, essentially represent an object as a flat record structure, consisting of a finite number of fields. Each field contains a fixed-length printable value. Thus, when manipulating a traditional database, an application program typically processes a large number of identically structured records and performs a similar manipulation on each record. An example might be deducting federal tax from each of perhaps thousands of employee payroll records. It might be necessary to relate other information to each record, and if so, logical or physical pointers are used. For example, occasionally an employee may have a partial tax exempt status; in these cases, the payroll record might point to a record in another file which tells the payroll system how to calculate that person's tax.

There are two very important distinctions that separate traditional business applications and engineering database applications. First, the objects being manipulated are typically not as easily represented as simple, flat records. And second, an engineering database does not usually have the very low schema to data ratio that is commonly found in business applications. This means that there are fewer objects of a given type. These two distinctions cause the record-oriented batch mode of business databases to not be as useful in an engineering application.

Specifically, VLSI and PCB designs, CAD/CAM objects, and software objects are often very intricate. These applications have unusual data modeling needs, such as the ability to allow an object to have structured or complex attributes, and the ability to represent unusual forms of data, such as derived data [20] and versions [28]. Further, when manipulating engineering databases, a user

often deals with a smaller number of much more intricate objects. Commonly, a business user would manipulate at one time many employees in a payroll database or savings accounts in a bank, whereas an engineer would manipulate only a few modules in a software system at one time. The engineer might also need to keep track of multiple versions of one program. In order to support the data modeling needs of a software environment, Cactis has drawn on two areas of recent database research, semantic and object-oriented data modeling.

1) Constructed Types: A Cactis database is viewed as a collection of *objects* similar to the objects found in Smalltalk [17]. Each data object has a group of data values attached to it called *attributes*. An attribute may have a value of any C data type (except pointer types). The set of attributes attached to an object is determined by the type of the object. The type of an object may be statically determined, or may vary dynamically on the basis of a predicate that defines subtype membership. As the attribute values of an object change, the object may meet or fail to meet the criteria for inclusion in a particular subtype and hence be moved up and/or down in the type hierarchy.

In addition to an internal structure defined by attribute values objects may be connected recursively by typed and directed *relationships* to form higher level or *abstract* objects. Thus, Cactis is a semantic model, similar to the entity-relationship model [7] and the semantic data model [18]. It is actually based on the insyde model [30]. In semantic modeling terminology [24], relationships are used to construct *aggregations*, or complex objects. Unlike conventional record-oriented database objects, aggregations may have properties (relationships) which are not printable.

An example type would be a Load_Module as shown by example data objects in Fig. 1. Simplistically, a Load_Module might be modeled as consisting of objects with one multivalued relationship, called Modules, which relates objects of type Load_Module to objects of type Module. Modules themselves would be aggregations related to other Modules. Load_Module might also have three attributes, one giving its name, another the date it was last formed, and a third giving the name of a file containing the actual object code. An example subtype might be Recent_Load_Modules, which would include of all load modules less than, say a week old.

In addition to the subtyping and aggregation capabilities, it is important to point out a fundamental difference between the Cactis data model and conventional models. This difference concerns the nature of relationships. In a conventional model the type of a relationship uniquely defines the types of the objects which are related. In the Cactis data model this is not true. A relationship only defines the type, direction, and number of values that flow between objects (see the next section on local behavior). Consequently, the exact type of the object at the other end of a relationship is not known until the relationship is traversed. This allows the types of objects to change or be extended dynamically without affecting related objects. The details of how an object is implemented, such as its exact type, its storage structure, or how it derives data, are all encapsulated within the object and hidden from the objects related to it. For example, in Fig. 1 we could transparently substitute an object of type New_Module for an object of type Module so long as the same external interface is maintained (i.e., the same connecting relationship type is used). This means that an object can be transparently related to an object of a type that did not even exist then the object was created. This feature is crucial for dynamically evolving systems like software environments.

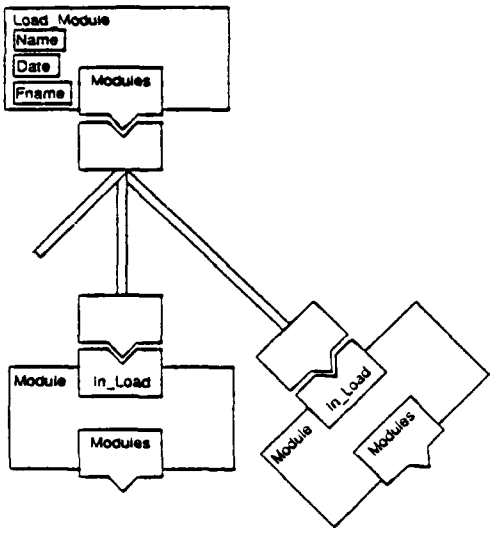


Fig. 1. Example objects of type Load_Module and Module.

2) Local Behavior: The Cactis data model is also an object-oriented model, in the behavioral sense. This means that individual objects in the database have embedded within them the means to respond to changes elsewhere in the database.

In a Cactis database, each object is an encapsulation of data along with a mechanism for implementing local behavior involving the data. The specific mechanism used in Cactis to support local behavior is that of *derived attributes*. This mechanism allows attributes to be derived by a function of other local attributes of the object as well as attribute values imported into the object over relationships. As a consequence, the interface to an object consists of its relationships and the values imported and exported across those relationships. An object can be seen as expecting certain types of values from its environment, and providing other values in return. The data language of Cactis [21] allows derivation functions to be constructed using all the arithmetic and logical operators found in the C language, plus several conditional and iteration constructs. Because of this flexibility, very complex behavior can be encapsulated in a Cactis object. The only restriction on these functions is that they may not have side effects (must be applicative). Examples of such behavior include such things as prioritizing bug report objects on the basis of deadlines derived from scheduling

object as discussed in Section III-A, or update of dataflow analysis information as described in Section III-B.

B. The Implementation of Cactis

In this section we focus on algorithms used by Cactis to manage derived data, as these are the features of the system which most heavily influence its ability to manage software databases. For further information about the Cactis implementation, see [23].

1) Derived Subtypes: In a Cactis database, the subtyping mechanism has the potential of creating a large amount of derived information. In order to prevent a storage problem from occurring, it is possible to allow Cactis to decide when a subtype should be materialized. The system supports an option which allows it to be self-adaptive, in that it continuously monitors itself to determine which subtypes should be reevaluated. Cactis will keep track of the usage history of each subtype, and then use this information to decide when to materialize a subtype. If this option is chosen, Cactis will only materialize a subtype under one of two conditions: if the subtype is queried, or if usage statistics indicate that it is cost-effective to materialize it. This allows the system to use less space and still provide quick access to certain subtypes.

Under this option, Cactis keeps an exponentially decaying average (by powers of 2) of the how many times each subtype has been referenced during the previous database sessions. This is multiplied by the number of objects which must be reevaluated for subtype inclusion (which can be derived from the object-level dependency information), in order to give a weighted factor. These weighted factors give an indication of which subtypes are highly volatile and are commonly referenced.

Subtypes with large weighted factors are materialized frequently, so that the system is able to anticipate query needs. This minimizes the delay involved in waiting for a response to examine a subtype, but does not materialize infrequently needed subtypes that are not volatile. The database designer may provide a parameter to decide how large a subtype's weighted cost factor must be to materialize it. This parameter is chosen with respect to the available storage. We note that this technique assumes a certain amount of locality of reference with respect to subtype examination.

2) Derived Attributes: The mechanism used to implement derived attributes is efficient in time and space. It is based loosely on recent work on incremental attribute evaluation [12], [43] and is philosophically similar to the mechanism used to maintain subtypes. In particular, the data managed by Cactis can be seen as an attributed graph. Such attributed graphs are a generalization of the attributed trees used for syntax directed and language specific editors [44]. Unfortunately, the optimal incremental update algorithm developed for this process [42] cannot be extended to attributed graphs. Instead a new incremental update algorithm has been developed for use with Cactis. Other algorithms for incremental update of attributed graphs are discussed in [1], [27]. However, these algorithms are either not suitable for use in a mass-storage environment, or do not handle arbitrary attributed graphs. Another system which relates derived attributes in a conventional attributed tree to a relations in a relational database is discussed in [19]. Finally, a system for standard incremental attribute evaluation using a variant of the standard optimal algorithm in a distributed environment is discussed in [26].

The Cactis update algorithm is optimal in the amortized set of attributes recomputed after a change, but somewhat less than optimal in total overhead. To be specific, when we amortize over any complete transaction sequence, the set of attribute reevaluations charged to a particular transaction is the set of attributes for which both of the following two conditions hold. 1) Either the computed value would change if evaluated, or the computed value of at least one attribute directly related to that attribute would change value (this is the minimum set of attributes to evaluate if all attributes must be updated after each transaction). 2) The attribute's value will eventually be used. That is, the attribute will be referenced at least once before either the end of the transaction sequence or the point where its value is overwritten by a different value. This is an amortized bound. Consequently, some transactions may perform more work, but as a consequence other transactions will perform less work. A complete analysis including complexity of overhead computations can be found in [22].

Because the algorithm is lazy, it can in fact outperform the optimal algorithm for trees in many cases. Unlike the standard optimal algorithm which assumes that all attributes must be up to date after each transaction, this algorithm only updates attributes which must be evaluated in order to ensure all declared constraints are met and all user requested values are available. The computation of attribute values which are not directly or indirectly observable from outside the system is deferred. This can result in very significant savings. Even without these savings, the algorithm vastly outperforms conventional trigger based systems [4], since its performance is never worse than linear, whereas triggers can exhibit exponential behavior in many cases. This linear behavior is crucial in applications such as software environments where long chains of attribute dependencies are required.

The Cactis incremental update algorithm works in two phases. The first phase determines what derived data is potentially affected by a change, and the second phase reevaluates the subset of that data which must actually be recomputed. During the first phase, data which might be directly or indirectly affected by a change is marked *out-of-date* using a traversal over a graph which represents the dependencies between attributes at the data level. During this traversal, certain attributes will be encountered which are designated *important*. These are the attributes which the system must ensure always have correct values. Whenever an important attribute is marked out-of date in the first phase of the algorithm, it is remembered for the second phase. The second phase of the algorithm recur-

sively recomputes the proper value of each important but out-of-date attribute based on the attribute evaluation rule attached to the attribute. Since the second phase operates in a demand driven way, it can be lazy. Consequently, it avoids evaluating attributes which are not actually needed. As a result, only the optimal set of attributes is actually recomputed after a change (although nonoptimal overhead is incurred to find these attributes).

In addition to efficient algorithmic techniques, the Cactis system also uses self-adaptive heuristic techniques to improve disk access performance over time. Statistics about past behavior are collected to be used as a predictor of future behavior. These statistics are used to cluster data which is frequently accessed. This results in significant performance gains. In addition, these statistics are used to schedule work to be done in an efficient manner. As described above, the incremental attribute evaluation algorithm is a pair of graph traversals. Because of the functional nature of the derivation rules used in Cactis, these traversals can proceed in a number of different orders. They could be done depth-first, breadth-first, or, as in Cactis, in an order which past behavior indicates will be efficient. In general the system schedules work which is expected to incur the least new disk accesses first. This preserves or frees buffer space for later work, thereby attempting to reduce disk accesses due to thrashing.

Extensive performance tests have been performed on Cactis [23]. These tests indicate the self-adaptive clustering and scheduling can save as much as 60 percent of disk accesses for databases that contain extensive amounts of derived data. Even for databases with little derived information, a 5 or 10 percent savings in disk accesses is usually realized.

III. Supporting Software Environments

In this section we describe on-going experiments involving Cactis. Various aspects of a software environment are being coded as Cactis applications. We are attempting to construct these tools in a uniform, integrated fashion. We feel that the data model supported by Cactis is a natural mechanism for specifying the sorts of data needed by a software environment. Our hope is that our experiments will serve three purposes. First, we hope to validate our claim. Second, we plan to illustrate that Cactis can support software environment data efficiently. And finally, we will evolve the implementation of our DBMS according to the experimental results we obtain. Clearly, we will discover significant ways in which Cactis can be improved.

Users of a software environment must perform many data manipulations that, unlike those of conventional database applications, involve complex derived data. Cactis provides a unified framework for manipulating derived information in a software environment at multiple levels of granularity. In this section, we illustrate this capability by discussing first derived data at the level of modules and whole programs, then at the level of statements or expressions within a program, and then finally, across multiple programs.

There have been numerous research projects aimed at supporting derived information in software environments. Notable examples are the work in version control [10], [40], [47], [49] and bug tracking [31]. Compared to these efforts, our research differs significantly in its goals. We are not primarily concerned with defining the functional requirements and developing the capabilities of these facilities. Rather, we are concerned with the ability of current database technology to provide useful support for such systems. In particular, we focus on the application of object-oriented data models to the support of software development, and on the efficient maintenance of disk-based data. In sum, our work is an attempt to provide effective, underlying database support for software engineering tools.

A. Programming in the Large

Objects used by the Cactis system are declared using the Cactis data language. In this language one may declare both object and relationship classes. A relationship class declaration provides names and types for the values that flow between related objects and indicates the direction of this flow. An object class provides names and types for a set of attribute values and provides the name, class, and direction of each potential relationship.

As a simple example, we might wish to create objects for use in a simple bug/fix tracking system. Fig. 2 shows the declarations we might use for such objects. Here several relationship classes have been defined. These include bug_fix, which will relate a bug_report to a fix_report, module_bug which will relate a bug_report to a module, and several others. Notice that, in order to give direction to relationships, one end of the relationship is called a *plug*, and the other a *socket*. Any object which possesses a plug of a given class may be related to any object which possesses a socket of that class. In some cases, relationships have been declared as *Multi Plug* or *Multi Socket*. This indicates a one-to-many relationship where multiple plugs (sockets) may attach to a single socket (plug). Relationships also allow values to flow between objects. In the case of a bug_fix relationship, one Boolean value (is_fixed) is transmitted from the plug end of the relationship to the socket end. In cases where a socket of class bug_fix is left unconnected, a value of false is transmitted by default.

A bug_report object can be related to a number of other objects as indicated by the relationships section of the declaration. A bug_report is related to the module which the bug occurs in by the in_module relationship. It may also be related to a fix_report via the fixed_by relationship, to a test result indicating the symptom via the symptom relationship, and finally, to objects representing project personnel via the reported_by and assigned_to relationships. Finally, a bug_report object has two attributes: report_words which contains a textual description

```
Relationship Class bug_fix
  Transmits
    is_fixed :      boolean To Socket, Default = False;
End Relationship;

Relationship Class module_bug Multi Plug End Relationship;
Relationship Class to_test_result Multi Plug End Relationship;
Relationship Class to_person Multi Plug End Relationship;
Relationship Class work_to_person Multi Socket End Relationship;

Object Class bug_report
  Relationships
    in_module      : module_bug Plug;
    symptom        : to_test_result Plug;
    fixed_by       : big_fix Socket;
    reported_by    : to_person Plug;
    assigned_to    : work_to_person Socket;
  Attributes
    report_words   : text_string;
    is_fixed       : boolean;
  Rules
    is_fixed := fixed_by.is_fixed;
End Object;

Object Class fix_report
  Relationships
    fixes          : bug_fix Plug;
    fixed_by       : to_person Plug;
    change         : to_delta Plug;
  Attributes
    report_words   : text_string;
  Rules
    fixes.is_fixed := True;
End Object;
```

Fig. 2. Objects for bug/fix tracking system.

of the bug, and is_fixed which indicates if the bug has been corrected.

Even though the definitions given in Fig. 2 are oversimplified, they can be used to illustrate several important capabilities of the Cactis system. First, we can see from this example that diverse but interelated data from different aspects of the development process can be unified in a single data model. In our example, three different kinds of data, management data involving project personnel, data representing actual program code, and data representing test results, have been integrated in order to implement a fourth aspect of the environment, a bug/fix tracking system.

Second, we note that the Cactis system is able to deal with large or unstructured types maintained by external tools. For example, the report_words attributes of both bug_reports and fix_reports is declared to be of type text_string. Because most of the details of type implementation are hidden from the system, this type could be implemented as an internal identifier which can be used by an external file-based string package to retrieve a large string from a special string file. Only the evaluation rules that deal with this type need be aware of its actual implementation. This ability of integrating private data managed by other tools is very important to the flexibility of a software environment. In particular, this capability is essential for dealing with object files or load modules. This data typically must be maintained and controlled by host operating system tools such as linkers and loaders.

Finally, we can see how derived data is used, and how the extensibility and subtyping capabilities of the system enhance its utility. The example uses derived data in a very limited way by computing the is_fixed attribute of objects of type bug_report. In this case, a fix_report object transmits this value across a bug_fix relationship to a bug_report object. When no such relationship exists, a value of false is supplied by default.

A more interesting, but still simple, example of derived data is given in Fig. 3. Here we have extended our example to incorporate scheduling information. A system of objects is used to schedule work to be done on the basis of milestones. A milestone object is given a target completion date and derives an expected completion data from the expected completion date of other milestones it depends on, along with an estimate of time required for work on the modules local to the milestone. As a consequence, a milestone object may automatically derive an attribute which indicates how late it is expected to be. This value can then be used to derive a priority to be placed on work for each module (or optionally this could be done manually with a scheduling tool). This priority can then be transmitted along the to_pri_module relationship defined in Fig. 3 to a pri_bug_report object.

Note that the pri_bug_report class is a subtype of bug_report. This indicates that it inherits all the relationships and attributes of bug_report. In addition, we have added a new relationship and two new attributes. The severity attribute now allows the bug to be weighted according to how severe it is. The bug_priority attribute computes a priority for the bug on the basis of the priority computed for (or assigned to) the relevant module along with the severity of the bug. Note that this computation can be done in a lazy fashion. If the bug has already been fixed, a priority of 0 is automatically defined, and no values need to compute a module priority are requested or recomputed. This is an example of where a computation involving a long chain of dependencies can be handled efficiently and automatically when needed, but is avoided when it is not needed. As an additional feature, the value assigned to the bug_priority attribute could also be used to define inclusion in a subtype. For example, one could define the subtype hot_bug as being all objects of the pri_bug_report class which have a bug_priority greater than some threshold. The use of this kind of derived subtype could make it easier to retrieve specific information regarding trouble spots in an ongoing project.

An important feature of the Cactis system is extensibility. The addition of new functionality to an object does not require that existing tools be modified. For example, any tool which uses a bug_report object can also transparently use a pri_bug_report object. The object oriented nature of the Cactis data model allows the implementation details of an object to be hidden, and allows a compatible external interface to be retained when classes are extended. Further, since relationships are defined on the basis of the values transmitted into and out of objects rather than on the basis of object classes themselves, a single object can be replaced by a group of related objects. One must only provide a compatible set of plugs and sockets.

```
Object Class milestone
  Relationships
    depends_on : milestone_dep Plug;
    ...
  Attributes
    target_compl, local_work, exp_compl, lateness : time;
  Rules
    exp_compl := ... ;
    lateness := exp_compl - target_compl;
    ...
End Object;

Relationship Class to_pn_module
  Transmits
    module_pnonty : sched_pnonty To Plug, Default = 0;
  Multi Plug
End Relationship;

Object Class pn_bug_report
  Subtype of bug_report;
  Relationships
    in_pn_mod : to_pn_module Plug;
  Attributes
    bug_pnonty : sched_pnonty;
    seventy    : bug_seventy;
  Rules
    bug_pnonty
      := If is_fixed Then 0
         Else assign_pnonty(in_pn_mod.module_pnonty, seventy);
End Object;
```

Fig. 3. Extended bug/fix tracking objects.

B. Programming in the Small

In the previous section, examples of manipulating derived data at the level of modules and at the level of a schedule for a whole project were considered. One of the important features of the Cactis data model is that it can also deal with very fine grained structures using the very same mechanism.

To illustrate this, Figs. 4, 5, and 6 define a series of objects for representing programs as abstract syntax trees. These sorts of structures are typical of the tree structures used as intermediate code for a compiler [48] or for representing programs in a syntax directed editor [46]. In this case we have modeled a simple language containing sequences of assignments, while loops, and if–then–else statements. These are modeled by the object classes assign_stmt, while_stmt, if_stmt, sequence, and empty_stmt. In addition, a number of object classes for representing expressions and other constructs would be required but are not shown. These objects can be combined to form trees using the stmt_dataflow and expr_dataflow relationships shown in Fig. 4.

Going beyond objects which simply represent programs, we have also included attributes and evaluation rules which can derive important information about the program automatically (these object definitions are adapted from an attribute grammar given in [14]). In this case we compute dataflow information which indicates the liveness of variables in the program. A variable is said to be *live* at a given program point if its value could be used at some point later in the program for some potential execution of the program. Variables whose value at a given point in a program cannot be used later in the program are said to be *dead*. This kind of information can be used to detect potential errors in a program or procedure [3], [16], [38]. For example, if a local variable of a procedure is live at the start of the procedure, then there is a potential execution path along which the variable may be used before it is assigned a value. This indicates a potential anomaly in the code. This anomaly information can then be used by other parts of the system to derive other information. However, the details of how this information is used is hidden. Consequently, it is possible to add new objects or new tools which use this information without reimplementing existing objects.

```
Relationship Class stmt_dataflow
  Transmits
    liveout   varset To Plug;   /* vars live on exit from stmt  */
    livein    varset To Socket; /* vars live on entry to stmt   */
    use       varset To Socket; /* vars used before set in stmt */
    thru      varset To Socket; /* vars not set on some path through stmt */
End Relationship;

Relationship Class expr_dataflow
  Transmits
    use : varset To Socket; /* vars used in expr */
End Relationship;


Object Class df_obj /* prototype for object using dataflow */
  Relationships
    parent : stmt_dataflow Plug;
  Attributes
    LIVEIN    varset; /* vars live on entry to stmt */
    LIVEOUT   varset; /* vars live on exit from stmt */
  Rules
    LIVEOUT := parent.liveout;
    parent.livein := LIVEIN;
End Object;
```

Fig. 4. Declarations for liveness analysis.

```
Object Class assign_stmt
  /* <stmt> ::= ID ":=" <expr> */
  Subtype of df_obj;
  Relationships
    asn_expr : expr_dataflow Socket; /* expression assigned */
  Attributes
    id : vand; /* var assigned to */
  Rules
    LIVEIN := (LIVEOUT - {id}) ∪ asn_expr.use;
    parent.use := asn_expr.use;
    parent.thru := all_vars - {id};
End Object;

Object Class if_stmt
  /* <stmt> ::= IF <expr> THEN <stmt_list> ELSE <stmt_list> END */
  Subtype of df_obj;
  Relationships
    cond_expr : expr_dataflow Socket; /* conditional expr */
    stmt1     : stmt_dataflow Socket; /* then clause      */
    stmt2     : stmt_dataflow Socket; /* else clause      */
  Rules
    LIVEIN := cond_expr.use ∪ stmt1.livein ∪ stmt2.livein;
    parent.use := cond_expr.use ∪ stmt1.use ∪ stmt2.use;
    parent.thru := stmt1.thru ∪ stmt2.thru;
    stmt1.liveout := LIVEOUT;
    stmt2.liveout := LIVEOUT;
End Object;

Object Class while_stmt
  /* <stmt> ::= WHILE <expr> DO <stmt_list> END */
  Subtype of df_obj;
  Relationships
    cond_expr : expr_dataflow Socket; /* conditional expr */
    stmt1     : stmt_dataflow Socket; /* loop body       */
  Rules
    LIVEIN := cond_expr.use ∪ stmt1.livein ∪ LIVEOUT;
    parent.use := cond_expr.use ∪ stmt1.use;
    parent.thru := all_vars;
    stmt1.liveout := cond_expr.use ∪ stmt1.use ∪ LIVEOUT;
End Object;
```

Fig. 5. Classes for computing liveness (part 1).

```
Object Class sequence
  /* <stmt_list> ::= <stmt> ";" <stmt_list> */
  Subtype of df_obj;
  Relationships
    stmt1      : stmt_dataflow Socket; /* first of sequence */
    stmt2      : stmt_dataflow Socket; /* rest of sequence */
  Rules
    LIVEIN := stmt1.livein;
    parent.use := stmt1.use ∪ (stmt2.use ∩ stmt1.thru);
    parent.thru := stmt1.thru ∪ stmt2.thru;
    stmt1.liveout := stmt2.livein;
    stmt2.liveout := LIVEOUT;
End Object;

Object Class empty_stmt
  /* <stmt_list> ::= EMPTY */
  Subtype of df_obj;
  Rules
    LIVEIN := LIVEOUT;
    parent.use := empty_set;
    parent.thru := all_vars;
End Object;
```

Fig. 6. Classes for computing liveness (part 2).

In order to compute liveness information, we introduce two attributes, *LIVEIN* and *LIVEOUT* which represent the set of variables which are live on entry to and exit from a statement, respectively. These attributes are defined as a part of the df_obj class given in Fig. 4. In addition, we also compute but do not store the values of two other sets: *use* which indicates the variables used before they are reassigned in a statement, and *thru* which indicates the set of variables which are not assigned along some potential path through the statement. These sets are transmitted between objects using the *stmt_dataflow* relationship for statements and the *expr_dataflow* relationship for expressions (here we assume expressions have no side effects).

The liveness computation is an example of a *backward* dataflow problem. That is, it proceeds in the direction opposite to control flow. In this case we use an initial value of LIVEOUT at the end of a procedure to calculate the value of LIVEIN at the beginning of the procedure along with LIVEOUT and LIVEIN at each point in between. For example, the variables live before an assignment statement include all the variables used in the expression being assigned, along with all the variables live after the assignment except the variable assigned to. Translating this into set notation we obtain the assign_stmt evaluation rule given in Fig. 5:

$$\text{LIVEIN} := (\text{LIVEOUT} - \{\text{id}\}) \cup \text{asn\_expr.use};$$

Similarly, the values live before a while loop include those used in the loop test expression and those live at the start of the body of the loop, as well as all of those live after the loop (since the loop may execute zero times). In set notation:

$$\text{LIVEIN} := \text{cond\_expr.use} \cup \text{stmt1.livein} \cup \text{LIVEOUT};$$

In the examples, we have used set notation for clarity. In the real specification used by the Cactis system this would be replaced by function calls implementing set operations. In general, the rules given compute the thru a use sets on the basis of their children and local information, then, given a value for LIVEOUT from their parer are able to compute the value of LIVEIN. The exact (dering of these computations is only partially define Within this partial order, computations are conceptual performed concurrently. At present, the computations a performed in an order expected to minimize disk acces The system is also being extended so that these comp tations can actually be performed in parallel as discuss in Section IV.

The computations we have defined in Figs. 4, 5, and do not involve cyclic dependencies. However, most d taflow problems are more easily characterized as the s lution to a fixed point problem and hence involve cycl attribute dependencies. The solution of such fixed poi problems using attribute grammars is considered in [14 In this work, an appropriate class of circular, but we defined attribute grammars is defined, and simple con tions are stated for guaranteeing termination of the r sulting cyclic attribute computations. Roughly speakin these conditions require that the evaluation functions volved be monotonic, and that the attributes involv come from finite domains. Under these conditions, a le fixed point solution can always be found by successiv approximation (i.e., iteration until convergence). The conditions extend straightforwardly to the attribut graphs used by the Cactis system.

The second phase of the Cactis incremental update gorithm can be modified slightly to handle cyclic but we defined attribute systems. While a value is being reev uated in the evaluation phase, it is given a special *in-pro ress* mark. If such a mark is encountered during evalu tion, a cycle exists. Such marks are used to identify strongly connected components of the dependency grap These can then be used to effectively compute an iterati solution to the fixed point problem. A related algorith for incremental evaluation of fixed points in convention attribute grammars is also studied in [25]. However, th algorithm is based on the optimal update algorithm t trees and does not extend to attributed graphs.

It should be noted that the Cactis system may not pe form well enough in some situations to support all aspec of programming in the small in a practical manner. In pa ticular, it is currently unknown if interactive edit-time s mantic tests would really be practical. If related progra components are spread across many disk blocks and termixed with other objects, the time to simply fetch the required disk blocks would be too long in itself. Hov ever, if the objects in question were placed together disk blocks, and their total size was close to that whi would fit in memory, adequate performance could pro ably be achieved. We are currently exploring mechanis for clustering of data which should allow good perf mance for programming in the small problems. Howeve only further experiments with the system will indica whether the system will actually be fast enough to supp this kind of problem.

C. Version Retrieval

As we have seen in the two previous subsections, the mechanisms of Cactis may be used to manipulate derived data at both the level of single programs and at the level statements or expression within a program. In a software environment, it is also necessary to manage groups of programs and metaobjects which are responsible for managing and organizing other objects. There are several contexts in which this is done. For example, an engineer may need to differentiate different *versions* of a related set of modules. The Cactis data model allows objects to be created which group together such a set of modules and form the basis for a version control tool which would allow groups of modules to be checked out, modified and checked in as a new version. In such a system, different versions of a program are normally not explicitly stored, but rather derived from a current version through some *delta* mechanism. In the Cactis system the delta information needed to recover old versions can be compact, and can itself be modeled as a set of objects. Because of the nature of the data model used, the delta mechanism can also be efficient and straightforward to implement.

Because a single change to a Cactis data object can cause derivations of new data arbitrarily far from the point of direct change, it seems that a delta mechanism would be difficult to implement. This is not the case. To understand why a Cactis supported delta mechanism can in fact be very simple, we can make a simple observation. Because all attribute evaluation rules are functional in nature, if a user modifies an object by changing an attribute value, the entire system can be restored to its original state simply by restoring the old value of the attribute. The same mechanism which automatically derives new data based on changes can also be used to automatically underive those changes. This observation also extends to structural changes as well as multiple changes made together. Even though a single change may have wide ranging effects in a database, only the data directly changed by the user needs to be stored in order to reverse those effects. This allows a very straightforward undo mechanism that can be used to reverse changes within a session, regardless of how the changes were done. Consequently, one need not build an undo mechanism into each tool, but can use the general mechanism provided by the system. In addition, this same capability can also provide an efficient delta mechanism by keeping objects representing edit operations and old values.

IV. Directions

Cactis is an operational, multiuser DBMS. It consists of about 70 000 lines of C code, and runs in the Berkeley UNIX environment. The system currently only provides centralized storage, but supports concurrent access by multiple users via a timestamping concurrency control technique.

The system is currently being extended from the current centralized implementation to a distributed implementation suitable for use in a distributed workstation environment. Properties of the Cactis data model make this transition particularly easy. Recall that the system only defines a partial order on computations and already uses an unpredictable evaluation order. To distribute the system, it is easy to (conceptually) insert a pair of special communication objects between any two related objects as shown in Fig. 7. Because the interface between two related objects consists solely of the type and number of values transmitted across the relationship, the actual transmission medium, whether local to a single machine or across a network, is transparent. The only modification to the sequential evaluation algorithm that is needed is synchronization between the first and second phases. The entire first phase must be complete before any computations in the second phase may begin. The existing concurrency control system will work in both the centralized and distributed systems.

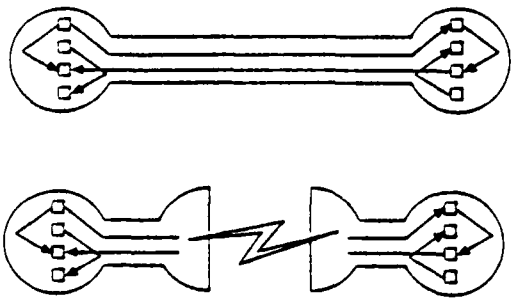


Fig. 7. Placing data on different hosts.

A further advantage of the Cactis data model is that it will be able to automatically manage replication of data in the distributed environment. With some small changes to the incremental update algorithm, replicated copies of data can be treated as a form of derived data. Since the incremental update algorithm is lazy, this system will amount to lazy replacement of replicated copies, with all book keeping and coordination handled automatically by the evaluation system.

Finally, due to the large, complex nature of derived data found in software environments, a software database must support unusual forms of transaction specification. Another direction of current research is support for these unusual requirements—particularly support for long and nested transactions. Typically, a designer will checkout a group of modules, work on them, and check them back in. This may entail a long interactive database transaction, during which the designer works on local versions of the modules. They would then be checked back in as new versions, and modules and object code may be formed which use them. Clearly, database transactions in a software environment are typically longer than standard, business-oriented transactions. They are also more complex, tending to spawn subtransactions which perform subtasks, such as compiling a program or relinking a load module. Software environment databases therefore require very dependable rollback mechanisms. Again, the simple mechanisms described in the last section for undo can be used to perform this rollback.

Long transactions may also necessitate more powerful and flexible constraint mechanisms than are typically found in conventional database systems. Constraints can themselves be described as a form of derived data. In a software application, designers may want the capability to specify complex constraints. An example would be that the versions which make up a load module be compatible in terms of the parameters they pass. Such constraints can easily be computed in the Cactis system as Boolean valued attributes. This form of attribute would use a predicate representing the constraint as its evaluation rule. The actual enforcement of constraints in the system can be flexible. Constraints may be tied to the termination of a transaction or the checkin point of a group of modules. In order to give a software designer tight control over the accuracy of a long, interactive transaction, constraints may also be enforced at the user's request or checked continuously.

There are a number of features of Cactis which are not completed. At this point, nested transactions are not supported, and cyclic attribute computations are not handled. Also, an effective software environment must support an interactive merge facility, whereby two versions of a module which were derived in parallel may be merged into one consistent module. Database support is clearly needed for this facility. Also, Cactis does not yet support incremental, run-time schema changes. This is likely to be a very serious shortcoming. An interesting question is whether—as many have suggested—the traditional distinction between the database administrators and end-users is not valid in a software environment database. In business applications, database administrators make schema changes in the process of designing and building the database using the DBMS, while end-users never make schema changes. However, in a softwave environment, it may be true that end-users make significant schema changes at run-time, by performing such operations as introducing new tools to a system under construction or by asking for new forms of consistency checks and constraint tests.

Currently, Cactis is being distributed to a small number of users. In order to evolve and fine tune the implementation of Cactis, plans call for instrumenting the system to gather statistics concerning real life software environment applications. Obviously, the general properties of software databases will vary from application to application, but at this point in time, very little is known about the characteristics of actual data. Thus, even very rudimentary information will be useful.

The sorts of information that we will collect include the following: the size of software databases, the ratio of schema to data size, and the number of objects in typical types. This will give us a feeling for the storage requirements of software environments. An interesting question is whether or not typical transactions will retrieve most of the data they need early (in the checkout phase), use it, then check it back in. This would reduce the problem of concurrency control. It would also indicate that techniques developed for main memory databases [34] m: prove useful. Information about the depth and breadth dependency graphs among derived attributes, the patte of repetition among database sessions, and the percent of set-oriented versus non-set-oriented database op tions will help us determine the effectiveness of our c tering and scheduling algorithms.

Acknowledgment

The authors would like to thank the team that has b constructing the software described in this paper: Drew, S. Gamalel-din, J. Jacobs, D. Lancaster, C. M ers, L. Neuberger, E. Pattern, D. Ravenscroft, T. R man, K. Rivard, J. Thomas, and G. Vanderlinden. In p ticular, P. Drew and T. Rebman deserve much credit constructing the physical access level of Cactis, and Thomas constructed the data language processor.

We would also like to thank L. Osterweil; many of ideas in this paper emerged as a result of conversati with him.

References

[1] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck, "In mental evaluation of attributed graphs," IBM Res. Rep. RC 13 Oct. 1987.

[2] P. Bernstein, "Database system support for software engineerin Wang Inst. Grad. Studies, Tech. Rep. 87-01, Feb. 1987.

[3] J. C. Browne and D. B. Johnson, "Fast: A second generation p gram analysis system," in *Proc. 3rd Int. Conf. Software Eng.*, 1978, pp. 142-148.

[4] O. P. Buneman and E. K. Clemons, "Efficiently monitoring re tional databases," *ACM Trans. Database Syst.*, vol. 4, pp. 368-3 Sept. 1979.

[5] R. H. Campbell and P. A. Kirslis, "The SAGA project: A system software development," in *Proc. Symp. Practical Software De opment Environments*, Pittsburgh, PA, 1985, pp. 73-80.

[6] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "O ject and file management in the EXODUS extensible database s tem," in *Proc. Twelfth Int. Conf. Very Large Databases*, Aug. 19 pp. 91-100.

[7] P. P. Chen, "The entity-relationship model—Toward a unified v of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9-36, 19

[8] G. M. Clemm, "Odin: An extensible software environment, rep and users reference manual," Univ. Colorado at Boulder, Tech. R 262-84, Mar. 1984.

[9] K. Cooper, K. Kennedy, and L. Torczon, "The impact of interp cedure analysis and optimization in the Rn programming envir ment," *ACM Trans. Program. Lang. Syst.*, pp. 491-523, Oct. 19

[10] I. D. Cottam, "The rigorous development of a system version con program," *IEEE Trans. Software Eng.*, vol. SE-10, no. 3, pp. 1 154, Mar. 1984.

[11] F. DeRemer and H. Kron, "Programming-in-the-large versus p gramming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, 80-86, June 1976.

[12] A. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation attribute grammars with application to syntax directed editors," *Conf. Rec. 8th Annu. ACM Symp. Principles of Programming L guages*, Jan. 1981, pp. 105-116.

[13] K. Dittrich and U. Dayal, *Int. Workshop Object-Oriented Databas* Pacific Grove, CA, Sept. 23-26, 1986.

[14] R. Farrow, "Automatic generation of fixed-point-finding evaluat for circular, but well-defined attribute grammars," *SIGPLAN tices*, vol. 21, pp. 85-98, July 1986.

[15] S. I. Feldman, "Make—A program for maintaining computer p grams," *Software—Practice and Experience*, vol. 9, pp. 255-2 Apr. 1979.

[16] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in softw reliability," *ACM Computing Surveys*, vol. 8, pp. 305-330, Se 1976.

[17] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley, 1983.
[18] M. Hammer and D. McLeod, "Database description with SDM: A semantic database model," *ACM Trans. Database Syst.*, vol. 6, no. 3, pp. 351–386, 1981.
[19] S. Horwitz and T. Teitelbaum, "Generating editing environments based on relations and attributes," *ACM Trans. Prog. Lang. Syst.*, vol. 8, pp. 577–608, Oct. 1986.
[20] S. Hudson and R. King, "CACTIS: A database system for specifying functionally-defined data," in *Proc. Workshop Object-Oriented Databases*, Pacific Grove, CA, Sept. 23–26, 1986, pp. 26–37.
[21] S. E. Hudson and R. King, "Object-oriented database support for software environments," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Francisco, CA, May 1987, pp. 491–503.
[22] S. E. Hudson, "Incremental attribute evaluation: An algorithm for lazy evaluation in graphs," Univ. of Arizona, Tech. Rep. 87-20, Aug. 1987.
[23] S. Hudson and R. King, "Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system," *ACM Trans. Database Syst.*, to be published.
[24] R. Hull and R. King, "Semantic database modeling: Survey, applications, and research issues," *ACM Comput. Surveys*, to be published.
[25] L. G. Jones and J. Simon, "Hierarchical VLSI design systems based on attribute grammars," in *Conf. Rec. 13th Annu. ACM Symp. Principles of Programming Languages*, Jan. 1986, pp. 58–69.
[26] S. K. Kaplan and G. E. Kaiser, "Incremental Attribute evaluation in distributed language-based environments," in *Proc. 5th Annu. Symp. Principles of Distributed Computing*, Calgary, Aug. 11–13, 1986, pp. 121–130.
[27] S. M. Kaplan, "Incremental attribute evaluation on node-label controlled graphs," Univ. Illinois, Tech. Rep. UIUCDCS-R-87-1309, May 1987.
[28] R. Katz and T. Lehman, "Version modeling concepts for computer-aided design databases," in *Proc. ACM SIGMOD Conf.*, May 1986, pp. 379–386.
[29] R. King and D. McLeod, "Semantic database models," in *Database Design*, S. B. Yao, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1985.
[30] ——, "A methodology and tool for designing office information systems," *ACM Trans. Office Inform. Syst.*, Jan. 1985.
[31] D. B. Knudsen, A. Barofsky, and L. R. Satz, "A modification request control system," *Tutorial: Automated Tools for Software Engineering*, 1976.
[32] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory J.*, vol. 2, pp. 127–145, June 1968.
[33] ——, "Semantics of context-free languages: Correction," *Math. Syst. Theory J.*, vol. 5, pp. 95–96, Mar. 1971.
[34] T. Lehman and M. Carey, "Query processing in main memory database management systems," in *Proc. ACM SIGMOD Conf.*, May 1986, pp. 239–250.
[35] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an object-oriented DBMS," in *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, Sept. 29–Oct. 2, 1986, pp. 472–482.
[36] D. Maier and J. Stein, "Indexing in an object-oriented DBMS," in *Proc. First Int. Workshop Object-Oriented Database*, Pacific Grove, CA, Sept. 23–26, 1986, pp. 171–182.
[37] F. Manola and U. Dayal, "PDM: An object-oriented data model," in *Proc. Workshop Object-Oriented Databases*, Pacific Grove, CA, Sept. 23–26, 1986, pp. 18–25.
[38] L. Osterweil, "Using data flow tools in software engineering," in *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 237–263.
[39] M. H. Penedo, "Prototyping a project master data base for software engineering environments," in *Proc. Second Symp. Practical Software Environments*, Dec. 1986.
[40] D. E. Perry, "Version control in the Inscape environment," in *Proc. 9th Int. Conf. Software Engineering*, 1987, pp. 142–149.
[41] M. L. Powell and M. A. Linton, "Database support for programming environments," in *Proc. Int. ACM Conf. Management of Data*, 1983.
[42] T. Reps, "Optimal-time incremental semantic analysis for syntax-directed editors," in *Conf. Rec. 9th Annu. ACM Symp. Principles of Programming Languages*, Jan. 1982, pp. 169–176.
[43] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Trans. Prog. Lang. Syst.*, vol. 5, pp. 449–477, July 1983.
[44] T. W. Reps, *Generating Language-Based Environments*. Cambridge, MA: MIT Press, 1984.
[45] R. Taylor, "Arcadia: A software development environment research project," Dep. Inform. Comput. Sci., Univ. California at Irvine, Tech. Rep., Apr. 1986.
[46] T. Teitelbaum and T. Reps, "The Cornell program synthesizer: A syntax-directed programming environment," *Commun. ACM*, vol. 24, pp. 563–573, Sept. 1981.
[47] W. F. Tichy, "Design, implementation, and evaluation of a revision control system," in *Proc. 6th IEEE Int. Conf. Software Engineering*, Sept. 1982, pp. 58–67.
[48] W. M. Waite and G. Goos, *Compiler Construction*. New York: Springer-Verlag, 1984.
[49] J. F. H. Winkler, "Version control in families of large programs," in *Proc. 9th Int. Conf. Software Engineering*, 1987, pp. 150–161.
[50] D. Woelk, W. Kim, and W. Luther, "An object-oriented approach to multimedia databases," in *Proc. ACM SIGMOD Conf.*, May 1986, pp. 311–325.

Scott E. Hudson (S'83–M'86) received the B.S. and M.S. degrees in computer science from Arizona State University, Tempe, in 1982 and 1983, respectively, and the Ph.D. degree from the University of Colorado, Boulder, in 1986.

He is currently an Assistant Professor in the Department of Computer Science at the University of Arizona in Tucson. His research interests include graphical user interfaces, user interface management systems, object-oriented database systems, and programming environments.

Dr. Hudson is a member of the Association for Computing Machinery.

Roger King (S'81–M'82) received the Ph.D. degree in computer science from the University of Southern California, Los Angeles, in 1982.

Since then he has been an Assistant Professor in the Department of Computer Science at the University of Colorado in Boulder. His research efforts are directed largely toward database support for software environments, semantic and object-oriented database implementation, graphical interfaces to databases, the support of derived data in databases, distributed database implementation, the design of rule-based databases, forms-based office interfaces, and computers and law. His research has been supported by the NSF, ONR, the Naval Ocean Systems Center, the Department of Energy, IBM, AT&T, and USWest.

Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System

To appear in ACM Transactions on Database Systems

Scott E. Hudson

Department of Computer Science
University of Arizona
Tucson, Arizona 85721

Roger King

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Abstract

Cactis is an object-oriented, multi-user DBMS developed at the University of Colorado. The system supports functionally-defined data and uses techniques based on attributed graphs to optimize the maintenance of functionally-defined data.

The implementation is self-adaptive in that the physical organization and the update algorithms dynamically change in order to reduce disk access. The system is also concurrent. At any given time there are some number of computations that must be performed to bring the database up to date; these computations are scheduled independently and performed when the expected cost to do so is minimal. The DBMS runs in the Unix/C Sun workstation environment.

Cactis is designed to support applications which require rich data modeling capabilities and the ability to specify functionally-defined data, but which also demand good performance. Specifically, Cactis is intended for use in the support of such applications as VLSI and PCB design, and software environments.

1. Introduction

Cactis is an object-oriented DBMS which supports a wide class of derived information - data which is computed from functions. In Cactis, a database object can have both

This work was supported in part by ONR under contract number N00014-86-K-0054, in part by NSF under grant DMC-8505164, and in part by Hewlett Packard under the American Electronics Association Faculty Development Program.

attributes and integrity constraints which are functionally-defined. Thus, the system is useful for automatically maintaining data which would normally have to be derived by an application program. As an example, a business database might keep track of objects called widgets. Each of these objects might have an attribute which is a pricing function that derives the selling cost of a widget in terms of the cost of its parts and the labor to put it together. There might be a constraint that says one sort of widget can never cost more than twice another sort of widget. In each of these cases, the Cactis system would automatically recompute derived values whenever necessary, and enforce constraints whenever updates are made.

There are two issues involved in supporting functionally-defined data: the necessary conceptual modeling capabilities of the DBMS and the physical modeling techniques required to implement this model efficiently. In the case of the Cactis system, the conceptual and physical models used by the database are identical. Cactis uses an attributed graph formalism which generalizes attribute grammars [24, 25] and incremental attribute evaluation techniques [12, 35] used in compilers and syntax directed editors. In order to make the system effective in terms of minimizing I/O, it has been constructed using self-adaptive and concurrent techniques.

The Cactis data model provides a useful and simple formalism for describing derived information. This simple formalism has in turn lead to a simple and straightforward implementation strategy. Indeed the authors have found that this formalism allowed us to construct a large system very quickly. The system was built by a team of twelve students in about a year, with the majority of the code being written by three of them.

The authors further discovered that the physical data model used was conducive to a concurrent implementation. At any point in time, the database is viewed as containing some number of pending computations that must be performed in order to keep derived data up to date. These computations may be generated by a number of transactions executing concurrently or, more often, may be the result of multiple subcomputations needed for a single transaction. Cactis is also self-adaptive, in that the system responds to usage patterns in two ways. First, the Cactis scheduler dynamically selects the next pending computations to perform by deciding which one will provide the best expected performance. The criteria for selecting the next computation may be that the result is required by a user or that the data objects required to perform the update can be obtained with little I/O.

The second self-adaptive technique, which works hand-in-hand with the scheduler, is the clusterer. This subsystem is run off-line and periodically reblocks the database according to the way in which data is historically accessed. The clusterer places two objects near each other if one is commonly used to derive the other. Thus, when the scheduler selects the next pending computation, the clusterer will have already minimized the I/O effort involved in performing the update.

Cactis was designed with specific sorts of data in mind. For example, complex engineering data is seen as a natural application. In particular, Cactis has been studied as a foundation for the support of software environments. A software environment is an application which requires the management of highly interconnected data. Modern environments attempt to provide a facility for managing the design, construction, testing, use, and eventual reuse of software. One of the most important requirements of a

software environment is providing a central store for managing the myriad of objects which make up a software project and keeping these objects up to date in the face of the many changes made over the lifetime of a project.

A number of features of the Cactis system make it conducive to such applications. First, the system supports the construction of complex data and type/subtype hierarchies. This is necessary in order to cleanly model such things as programs, requirement and design specifications, progress and bug reports, configurations, and documentation which are representative of the data found in existing and proposed environments, as described in [11, 33]. Next, functionally-defined attributes are also very useful in a software environment application, as such a system might contain large amounts of derived data in the form of compilations, cost calculations, and scheduling dependencies. Cactis provides a mechanism for constructing derived data which, although it supports a smaller class of derived information than generalized triggers [6], is much more efficient than triggers. The system also allows the user to extend the type structure, which is useful for adding new tools to such a system without disturbing existing functionality.

The capability to support complex data and type/subtype hierarchies discussed above is provided by the subsystem of Cactis, called Sembase, a tool constructed at the University of Colorado (see [15, 22]). The other three capabilities form a major recent development effort. A brief preliminary report describing the design of Cactis appears in [18]. The system is now complete and consists of approximately 65,000 lines of C code, and uses a timestamping concurrency control technique.

2. Related Work

Recently, significant interest has developed in object-oriented database models, and in models which represent derived information. A large class of such models are commonly called semantic models. A complete discussion of such models and their relationship to traditional models may be found in [20, 23]. Briefly, traditional database models support record-like structures and/or inter-record links (e.g., the relational, hierarchical, and network models). Semantic models support expressive data relationships; a typical semantic model allows a designer to specify complex objects, and also supports at least one form of derived relationship, generalization (sometimes called subtyping). With generalization, one sort of object can be defined as belonging to a subcategory of a larger category of objects. Semantic models are limited in the sense that they commonly do not include support of methods which operate on data objects, as is typical in more generalized object-oriented models.

For a discussion of a number of research efforts directed at implementing object-oriented database systems, see [13]. Such systems vary from extensions to the relational model to handle complex data [41] to database implementations based on the message passing paradigm of Smalltalk [27, 28]. An object-oriented system which uses persistent programming techniques is described in [2]. [9, 38] discuss data structures and access methods used to implement semantic databases. Object-oriented implementations designed to support extensible databases are described in [3, 8]; these systems are toolkits which allow the user to tailor data modeling and storage mechanisms of a database system. Another extensible system, designed for such applications as engineering, is described in [29]. There have also been some work in the area of database support for

software engineering; see [31, 42, 45].

A common theme running through many of these projects is that an object-oriented system must be able to support a wide variety of objects and allow attributes of objects to be derived in terms of other data items in the database. Other researchers have stressed the importance of derived data in knowledge based databases [26, 30, 36]. Much of the previous work in this area has come from AI research oriented toward constraint based programming systems [5].

During the development of the Sembase subsystem of Cactis a couple lessons were learned. While this project did produce a system capable of supporting a wide class of object-oriented systems, including some forms of derived information, it fell short in two ways. First of all, only a subset of first order predicate calculus expressions may be used to manage derived data. Secondly, the code, while very efficient, is tricky and inelegant. Cactis supports a much wider class of derived information, and does so in a clean fashion, based on a simple algorithmic model.

In the next Section, the Cactis data model is briefly described, Section 4 considers how algorithmically efficient incremental update can be performed, and the implementation of Cactis is discussed in Section 5. Section 6 describes the Cactis data language; the examples are taken from a software environment application. Section 7 discusses performance tests which have been conducted. Finally, Section 8 considers some of the limitations of the system, details the directions this research is currently taking, and provides conclusions.

3. The Cactis Data Model

In this section we will first informally introduce and motivate the Cactis data model, and then introduce a more formal description along with an example.

Informally, the data in a Cactis database consists of a collection of typed objects. Each object represents some entity modeled in the database, and encapsulates both the data and the behavior associated with that entity. Objects contain internal (hidden) structure, and can be related to one another externally by relationships to create external structure. In conventional object-oriented systems such as Smalltalk [16], the external interface to an object is a set of messages which it can respond to. In a Cactis database the interface to an object is the set of values that flow into and out of the object across relationships.

A Cactis schema defines, for each object type, an internal implementation. This internal implementation defines the values stored within an object, and the constraints placed on these values. In addition, the schema specifies how these values may be functionally derived from values passed into the object across relationships, and how the object may derive the values to be passed out of the object across those relationships. This functional derivation of data values allows objects to respond to their environment. When a data value imported into an object across a relationship changes value, the internal implementation of the object may respond by recomputing local data, and by providing new data which is exported from the object across relationships. This automatic derivation of data implements the behavior of the object.

An important distinction between Cactis and other high-level models, such as the predominant semantic models, is that it handles relationships very differently. For exam-

ple, in the Entity-Relationship Model [10], the Semantic Data Model [17], and the Functional Data Model [21, 37], relationships are defined with types; an object type definition encompasses the relationships it participates in - including the range types of the given relationships. This is not true in Cactis, where relationships are typed separately, and the range type of a relationship does not depend on the domain type.

A further consequence is that a DBMS based on a semantic model is not conducive to schema restructuring. In order to vary the manner in which two types are connected via a relationship, one must redefine two types. In Cactis this can be done dynamically, at run-time, by merely assigning a new relationship to both sets of connectors.

To make these concepts and their implications more concrete, we now introduce a more formal definition of the model and provide an example in a graphical notation.

Each object in a Cactis database is an instance of a type from a hierarchical type system. Multiple inheritance is supported and resolved at schema definition time. The type of an object can be either explicitly declared, or chosen dynamically from a family of subtypes using predicates which are evaluated whenever updates are made. Each type specifies the internal implementation of a class of objects along with their external interface. The internal implementation of an object indicates a set of typed data values called *attributes* that are stored within the object. Currently, attributes may take values from any type definable in the C programming language.

In addition to attributes, an object's internal implementation specifies how some of these values may be functionally derived from other values, either within the same object, or imported across relationships. This specification comes in the form of *attribute evaluation rules* attached to attributes. An evaluation rule attached to an attribute

Figure 1. A Simplified Sample Schema

indicates that the attribute's (observable) value should always be equal to the expression given in the rule. Attributes which have an evaluation rule attached are called *derived* attributes, whereas values which are simply stored are called *intrinsic* attributes. Attribute evaluation rules are applicative and may not have side effects. Currently, attribute evaluation rules are expressed in a data language which is compiled into C, and can compute any function expressible in the C language. Finally, the internal implementation of an object may specify additional constraints on the attributes of an object. These con-

straints provide predicates that are to hold true after every database update. These constraints are implemented using normal attribute evaluation rules which return a boolean value which is the result of evaluating the constraint predicate.

The external interface to an object describes the relationships it may have with other objects and the way values flow across those relationships. Each relationship is typed and directed. Type and direction are used to represent semantic concepts. For example, a relationship might represent the semantic concept "component-of". Because of the need for directedness, relationships have two distinct ends which we will call *connectors*. In order to distinguish the ends of a of a relationship and hence establish its directedness, we say that each relationship possesses a *black* connector, and a *white* connector. The external interface for an object consist of a set of connectors which are intended to match the connectors of a relationship type.

Figure 2. Example Objects

As an example, Figure 1 uses a simplified graphical notation (similar to [1]) to show three object types: Proj_Cost, Comp_Cost, and Fixed_Cost, along with one relationship type: CST. (To simplify the graphical presentatio.. we have used only 1:1 relationships. A more realistic set of objects for this task would use 1:many relationships.) Proj_Cost objects represent and compute the total estimated cost for a software project as derived from other objects. A Comp_Cost object automatically computes the estimated cost of a part of a project on the basis of a formula encapsulated in the object, along with the estimates for subparts it is related to. Finally, a Fixed_Cost object represents an estimate which has been explicitly entered by a user.

Note that, in this example, objects of type Proj_Cost possesses a black CST connector, hence they may be related to any object which possesses a white CST connector, in this case objects of type Comp_Cost or Fixed_Cost. Similarly, objects of type Comp_Cost also possess a black CST connector, therefore they may be related to objects of type Comp_Cost or Fixed_Cost. An important point here is that an object of type Proj_Cost cannot know, and does not care, whether it is related to an object of type Comp_Cost, or type Fixed_Cost, or of some type not shown (perhaps even a type that did not exist when Proj_Cost was defined). An object of type Proj_Cost is only concerned with the values that flow across a CST relationship, not how they are provided. This allows a Cactis database to be both flexible and extensible while retaining strong typing. In this case both automatically computed and manually entered cost estimates can be handled uniformly, and a number of different cost estimating formulas or methods could all be handled uniformly and transparently.

To illustrate how actual objects of the types shown in Figure 1 might behave, Figure 2 provides a more detailed view of some sample objects of those types. Here we have shown not only relationships, but also internal attribute values, the values that flow across relationships, and the fact that some internal attributes are derived. In this case, a Comp_Cost object is shown to have three internal attributes: two intrinsic attributes, and, as shown by the arrows, a derived attribute which computes its value as a function (not shown) of the intrinsic attributes and the values passed into the object from the outside. Similarly, objects of type Proj_Cost have a single derived attribute, and objects of type Fixed_Cost have a single intrinsic attribute.

To see how a Cactis database handles updates, consider what would happen if the single attribute in the Fixed_Cost object shown were changed. Because of the specific data level relationships that have been established, changing the Fixed_Cost attribute indirectly affects both the derived attribute of the middle Comp_Cost object and the derived attribute of the Proj_Cost object. When such a change is made, the Cactis system is responsible for identifying and performing any indirect updates needed to bring the system up to date with respect to all attribute evaluation rules. In this case, the system must recompute two attributes. Later we will see that, if the results are not immediately needed, such computations are not performed immediately, but deferred until the value is actually required. The user also has the option of declaring that an attribute is *important*. This indicates that the system is to maintain the correct value for the attribute at all times. This capability is used, for example, to insure that the attributes computing the predicate for each constraint are always reevaluated if they could change.

In this section we have seen an outline of the Cactis data model. We have seen that data is modeled as a set of objects connected by relationships. These objects encapsulate data and behavior by providing evaluation rules that indicated how some data is derived in terms of other data. Looking at this model in a different light, it can be seen as equivalent to a form of attributed graph. That is, a graph which has its nodes decorated with attribute values, and for which there is a set of defining attribute evaluation rules which describe how attributes may be derived from other attributes. Consequently, this model is related to the attribute grammars used in compilers and language dependent editors. This relationship will point the way to a new incremental update algorithm in the next section.

4. Efficient Incremental Update

A number of data models have made provisions for functionally derived or active data which can respond to changes in surrounding data. As a typical example, the LOOPS object-oriented programming system [4] provides active data which can invoke a procedure whenever a data item is accessed. The implementations of active values in LOOPS, as well as most other systems which provide active or derived data use techniques equivalent to triggers [6] attached to data. While this method is adequate for sparsely interconnected data, it can present problems for more highly interconnected data. Since there is no restriction on the kinds of actions performed by triggers, the order of their firing can change their overall effect. While this allows triggers to be extremely flexible, it can also become very difficult to keep track of the interrelationships between triggers. Hence, it is easy for errors involving unforeseen interrelationships to occur, and much more difficult to predict the behavior of the system under unexpected cir-

cumstances.

By contrast, the effects of attribute evaluation computations used in the Cactis system are much easier to isolate and understand. Each data type in the system can be understood in terms of the the values it transmits out across relationships, the values it expects to receive across relationships, and the local attributes of the object. This allows the schema to be designed in a structured fashion and brings with it many of the advantages of modern structured programming techniques.

Even if we can adequately deal with the unconstrained and unstructured nature of triggers, they can also be highly inefficient. Figure 3 shows the interrelationships between several pieces of data. The arcs in the graph represent the fact that a change in one piece of data invokes a trigger which modifies another piece of data. For example, modifying the data marked A affects the data items marked B and C. If we choose a naive ordering for recomputing data values after a change, we may waste a great deal of work by computing the same data values several times. For example, a simple trigger mechanism might work recursively, invoking new triggers as soon as data changes.

Figure 3. A System of Dependencies

However, in our example, this simple scheme would result in recomputing data value J eight times, once for each path from the original change to the data item. In fact, only a few of the many possible orderings of computations does not needlessly recompute some data values. In this case, a breadth first order is optimal, however, it is easy to construct graphs where this is not true. Any trigger mechanism which uses a fixed ordering of some sort (e.g depth first or breadth first) will needlessly recompute some values for many graphs found in practice. In particular, any graph which contains dependencies which are not limited to trees or linear chains can cause this behavior. In the worst case triggers can needlessly recompute an exponential number of values. For example, if a depth first order is used on an extended graph of the type shown in Figure 3, $O(2^N)$ wasted computations will be performed. On the other hand, the attribute evaluation technique used in the Cactis system will not evaluate any attribute that is not actually needed, and will not evaluate any given attribute more than once.

Cactis supports a number of primitives which the data language described in Section 6 has been built on top of. These primitives include operations for creating and deleting objects, creating and deleting relationships between objects, defining predicates and subtypes, and primitives for retrieving and replacing attribute values.

Whenever these primitives are used to change a database, Cactis must ensure that all attribute values in the database retain a value which is consistent with the attribute rules of the system. This requires some sort of attribute evaluation strategy or algorithm. One approach would be to recompute all attribute values every time a change is made to any part of the system. This is clearly too expensive. What is needed is an algorithm for incremental attribute evaluation, which computes only those attributes whose values

change as a result of a given database modification. This problem also arises in the area of syntax directed editing systems, so it is not surprising that algorithms exist to solve this problem for the attribute grammars used in that application. The most successful of these algorithms is due to Reps [34]. Reps' algorithm is optimal in the sense that only attributes whose values actually change are recomputed.

Unfortunately, Reps' algorithm, while optimal for attributed trees, does not extend directly to the arbitrary graphs used by Cactis. Instead, a new incremental attribute evaluation algorithm has been designed for Cactis. This new algorithm exhibits performance which is similar to Reps' algorithm, but does have an inferior worst case upper bound on the amount of overhead incurred.

The algorithm works by using a strategy which first determines what work has to be done, then performs the actual computations. The algorithm uses the *dependencies* between attributes. An attribute is *dependent* on another attribute if that attribute is mentioned in its attribute evaluation rule (i.e. is needed to compute the derived value of that attribute). When the value of an intrinsic attribute is changed, it may cause the attributes which depend on it to become out of date with respect to their defining attribute evaluation rules. Instead of immediately recomputing these values, we simply mark them as *out-of-date*. We then find all attributes which are dependent on these newly out-of-date attributes, and mark them out-of-date as well.

This process continues until we have marked all affected attributes. During this process of marking, we determine if each marked attribute is *important*. Designating an attribute important indicates that the system is to maintain its correct value at all times. Important attributes include those explicitly designated as important by the user, and

those that compute constraint predicates. When we have completed marking attributes during the first phase of the algorithm, we will have obtained a list of attributes which are both out-of-date and important. We then use a demand driven algorithm to evaluate these attributes in a simple recursive manner. The calculation of attribute values which are not important may be deferred, as they have no immediate affect on the database. If the user explicitly requests the value of attributes (i.e. makes a query) new computations of out-of-date attributes may be invoked in order to obtain correct values. A similar implementation approach using lazy evaluation is described in [7].

We will now consider the efficiency of the attribute evaluation algorithm. We will do this analysis on the basis of changing a single attribute value, however the results extend to multiple attribute changes as well as to changes in the structure of the data graph.

To begin we will define several terms. The relationship of "depends on" defines a directed graph with attributes labeling nodes. This graph, which we will call **depends_on**, will have an edge from the node labeled with A to the node labeled with B if and only if attribute A depends on attribute B. We define the graph **Inverse**(g) as simply the graph g with all the edges reversed. We define a graph **Reachable**(n,g) as the subgraph of graph g which is connected to node n (that is which is reachable by following edges from node n.) Finally we define the set **Nodes**(g) to be all the nodes of a graph g, and the set **Edges**(g) to be all the edges of a graph g.

For a given attribute A we will define a graph **Could_Change**(A) which describes all the attributes that might change value if a new value is assigned to the attribute A. This graph contains the set of attributes that depend on A either directly or indirectly.

Specifically:

Could_Change(A) = **Reachable**(A, **Inverse**(depends_on))

In other words the subgraph reachable by following the depends on relationship backwards from A.

In addition to the **Could_Change** graph we can also define a set **Change**(A,V). This set will contain all attributes which must be reevaluated in order to insure that all affected attributes have a correct value. More precisely, the set **Change**(A,V) contains all attributes that either 1) require a new value after changing A to V, or 2) are directly derived from some attribute that requires a new value. Note that **Change**(A,V) $\subseteq$ **Nodes**(**Could_Change**(A)) for all attributes A and values V, and that **Change**(A,V) = { } when the current value of attribute A is already V.

When the value V is assigned to an attribute A in the Cactis attribute evaluation algorithm, no attributes other than those in the set **Change**(A,V) are reevaluated when we amortize over any transaction sequence[1]. This is exactly the same set that would have been reevaluated by Reps' optimal evaluation algorithm.

Reps' algorithm is also optimal in the total overhead incurred in attribute evaluation. Its total overhead in both best and worst cases is limited to O(| **Change**(A,V)|). However, Reps' algorithm uses some special properties of trees to achieve this result. These properties do not apply to the more general graphs used by the Cactis system

The overhead of the Cactis algorithm is not optimal. In particular its worst case amortized overhead is:

[1] A single change may recompute attributes outside the Change(A,V) set, however, the evaluation of all such attributes must have been deferred from the Change set of some previous transaction. Hence the evaluation of these attributes can be charged to the amortized cost function of that previous transaction.

$O(|\text{Nodes}(\text{Could\_Change}(A))| + |\text{Edges}(\text{Could\_Change}(A))|)$

This behavior comes from the mark out-of-date phase of the algorithm which does a depth first traversal starting from the node A. In the worst case this traversal may visit the entire Could_Change graph for A hence visiting each node and edge in this graph. However, this is the worst case behavior. In many real cases this traversal will be cut short by finding attributes which are already out-of-date. For example if an attribute A were assigned two different values in a row before updating the system, the second assignment would only update A and not visit any other attributes and hence incur only O(1) overhead. In general the actual performance of the Cactis attribute evaluation algorithm will depend on the attributes involved, particularly on whether some attributes may remain as out-of-date for long periods of time if they are not important and are not accessed. Also, if a given attribute is changed as a result of two different primitive updates to intrinsic attributes, the given attribute will only be reevaluated once (unless of course, the given attribute has been accessed or used to compute a constraint predicate before the second primitive update is performed).

In order to support the primitives which break and establish relationships, a process similar to that used for intrinsic attribute changes is used. When a relationship is broken, the system determines which derived attributes depend on values that are passed across the relationship. These attributes are marked out-of-date just as if an intrinsic attribute had changed. When a relationship is established, the second half of the attribute evaluation algorithm is invoked to evaluate attributes which are out-of-date and important. In order to ensure that derived attributes can always be given a valid value, the database ensures that connectors are not left dangling without a matching relationship. If relationships for each connector of an object are not explicitly provided by the transaction, the

system will automatically provide default values to replace any values what would normally flow into the object along the missing relationships. As a final note, please notice that the primitive to delete an object can be treated the same as breaking all relationships to the object, and the primitive to create an object does not affect attribute evaluation until relationships are established.

During the evaluation of attributes, certain attributes will compute constraint predicates. After such an attribute is evaluated, its value is tested. If it evaluates false, a constraint violation exists. Under user control, this can either causes the transaction invoking the evaluation to fail and be rolled back or, optionally, a special recovery action associated with the constraint can be invoked to attempt to recover from the violation. In either case, the constraint must be satisfied or the transaction invoking the evaluation will fail and be rolled back.

5. The Storage Structures and Access Methods of Cactis

We begin our discussion of the implementation of Cactis by describing a straight forward implementation of the model which does not attempt to optimize disk access. We will then show how Cactis uses a self-adaptive and concurrent approach to create an optimized implementation of the model.

In this section we concentrate on the implementation of the attribute evaluation portion of the Cactis model. This represents the vast majority of the code within Cactis. In the last section we gave an informal presentation of the incremental attribute evaluation algorithm used by the Cactis data model. In this section we give a more concrete and specific description.

The algorithm uses several pieces of information for each attribute in the data, including:

value - The actual value assigned to the attribute.

outofdate - A boolean value which indicates whether the attribute has been marked out-of-date.

changetime - An integer timestamp that indicates when the value was last assigned a new value.

readtime - An integer timestamp that indicates when the value was last used for a computation or read by the user.

In addition, attribute information that is the same for all data objects of one object type is stored in the schema. This information includes:

important - A boolean value which indicates if the attribute is designated important.

evalproc - A procedure which encodes the attribute evaluation rule for the attribute.

dependson - A set of things that this attribute depends on.

The **value** of the attribute is simply whatever value the attribute currently has. This value may or may not be correct depending on whether the attribute is up to date. The **outofdate** flag indicates whether the attribute might be out of date with respect to its defining attribute evaluation rule. When **outofdate** is false the attribute will have a correct value. When **outofdate** is true the attribute may or may not have a correct value. It is important to note that the database need not know anything about the internal structure of the attribute value other than its total size. Knowledge about all other aspects of

the attribute is encapsulated in the **evalproc** for the attribute. This encapsulation greatly simplifies the construction of the attribute evaluation system.

The **important** flag indicates if the attribute is to be considered important. If an attribute is considered important, the system will ensure that it always has an up to date value. In general, the **important** flag for an attribute within an object of a given type does not change, hence it is stored in the schema. Occasionally the importance of an attribute should change over time and depend on a predicate. In this case we can introduce a new attribute which is designated important. The evaluation rule for this new attribute will evaluate the predicate which determines if the original attribute is to be considered important. The evaluation rule will then request the correct value of the original attribute if and only if it is to be considered important (i.e., the predicate evaluates true).

The **changetime** associated with an attribute gives the timestamp for the last time the attribute value was changed. This **changetime** information can be used to avoid unnecessary attribute evaluations. If the **changetime** of an attribute being evaluated is later than the **changetimes** of all of the attributes it depends on, it need not be reevaluated nor its **changetime** modified. If an attribute is reevaluated but does not actually change value, its **changetime** can also be left undisturbed. This allows us to avoid chains of attribute evaluations which do not result in any value changes (although we must still incur overhead for these attributes). The **changetime** and **readtime** of an attribute are used in the concurrency control algorithm discussed in Section 5.4.

The **evalproc** of the attribute is a procedure which encodes the attribute's evaluation rule. This procedure will request other attribute values that it needs to perform its evaluation and will compute a new **value** and **changetime** for the attribute when needed.

This procedure is provided by the data definition language compiler on the basis of the attribute evaluation rule given by the user. Since the **evalproc** of an attribute of an object of a given type does not change, it is stored in the schema. The form of the **evalproc** is a compiled C function. This currently limits the extent to which dynamic schema changes can be made without relinking. An improvement which has not yet been implemented would be to represent an **evalproc** as a list of machine independent P-codes like those used to implement some machine dependent compilers [32]. In the later case, these P-codes would be interpreted rather than executed directly. This will allow new attributes and evaluation routines to be added while the database is running.

Finally, the **dependson** set associated with an attribute encodes those things (if any) that the attribute will need to compute its value. Again, since the **dependson** set of an attribute within an object of a particular type does not change, this information is stored in the schema. Further note that the dependson sets only encode the portion of the overall dependency graph that is local to a given object type. Traversal of the actual global dependency graph is performed by dynamically linking together the appropriate local dependency subgraphs found in the schema. The global dependency graph is never explicitly constructed or stored.

As detailed above, a Cactis database must store "overhead" information with each attribute of each data object. However, since the system is designed to support applications with large complex objects, the space for this information may not prove to be significant in practice. In particular, if 32 bit time stamps are used, this information only amounts to 65 bits of storage beyond that used by the attribute value itself.

5.1. A Naive Evaluation Algorithm

We will now discuss a simple version of the Cactis attribute evaluation algorithm which is efficient in the number of attributes which are evaluated, but which is naive with respect to I/O cost. In the next section we will consider an improved algorithm which attempts to optimize the amount of I/O performed. Recall that the attribute evaluation algorithm has two phases. An initial "mark out-of-date" phase, and a second phase which reevaluates attributes as needed. The first of these phases is initiated when attributes are assigned new values. The routines Set_Attr_Value and Mark_Out_Of_Date given in Figure 4 show how this phase is implemented.

Each time an attribute is assigned a new value, all attributes that depend on that attribute directly or indirectly are marked as out-of-date using the Mark_Out_Of_Date routine, and the changetime for the modified attribute is set to the current virtual time. An attribute is said to depend on another attribute if it might be needed to evaluate its value (in other words if it is mentioned in its attribute evaluation rule.) For example if the attribute A uses the following attribute evaluation rule from the schema:

A := (B + C) * (D - 2);

then the attribute A will depend on attributes B, C, and D.

The Mark_Out_Of_Date routine is a simple depth first marking procedure. The only special thing that it does is to record on an evaluation list any attribute which is designated important. Attributes on this list will be those that are currently both important and out-of-date. These are the attributes which the system will be concerned with in the second phase of attribute evaluation.

```
Procedure Set_Attr_Value(
        InOut A : Attribute;
        In V : Value;
        InOut Eval_List : List of Attribute)

  Begin
    Mark_Out_Of_Date(A, Eval_List);
    A.value := V;
    A.outofdate := FALSE;
    A.changetime := NOW;
  End;

Procedure Mark_Out_Of_Date(
        InOut A : Attribute;
        InOut Eval_List : List of Attributes)

  Begin
    If Not A.outofdate Then Begin
      If A.important Then
        Eval_List := Eval_List || A;
      A.outofdate := TRUE;
      For Each B such that B depends on A Do
        Mark_Out_Of_Date(B,Eval_List);
    End;
  End;
```

Figure 4. "Naive" Attribute Evaluation Routines

As shown here, the Mark_Out_Of_Date routine makes no attempt to minimize disk access costs, and always uses a fixed depth first order of traversal. In the next section we will consider how we can dynamically choose a traversal order which is most likely to be efficient in terms of disk access.

The second phase of attribute evaluation algorithm is invoked after a series of attribute changes using the Update_System routine given in Figure 5. This routine simply evaluates each of the attributes on the evaluation list that was collected in the mark out-of-date phase of attribute evaluation. Each attribute is evaluated by the Eval_Attr routine

```
Procedure Eval_Attr(InOut A : Attribute) : <Value, TimeStamp>

  Begin
    If A.outofdate Then
      A.evalproc(A);
    Return(<A.value, A.changetime>);
  End;

Procedure Update_System(InOut Eval_List : List of Attribute)
        Local Var Dummy : <Value,Timestamp>;

  Begin
    For Each A on Eval_List Do
      Dummy := Eval_Attr(A);
    Eval_List := Empty;
  End;
```

Figure 5. "Naive" Attribute Evaluation Routines

shown in Figure 5. This routine uses the evalproc attached to the attribute to reevaluate it if it is marked out-of-date, then returns the value and a timestamp indicating when the value was last changed.

The evalproc attached to each attribute encodes the attribute evaluation rule for the attribute (if any). These routines are slightly more complex than the attribute evaluation rule expressions they are derived from because they examine the change time from all the values they request in order to determine if the attribute computation can be avoided. However, they follow a very rigid pattern and are easily created by the data definition language compiler based on the attribute evaluation rules given by the user.

In addition to simple expressions, the attribute evaluation algorithm can handle conditional expressions. For example, we can have an attribute E whose attribute evaluation

rule is:

E := If B Then F Else G;

This rule is implemented using an evalproc which is "lazy". That is a routine which only requests values if they are actually needed. For example, when the attribute B happens to be true we request the value of attribute F but not attribute G. This laziness property allows out-of-date values to remain in the database until they are actually needed. This can represent a significant savings.

It is interesting to note that because of the clean formalism of attributed graphs that we have used, we can express each of the four central routines used for automatic update of values in under 15 lines of pseudo code.

In the next section we consider how to optimize update with respect to disk access. Again we will find that the formalism used allows the optimized update algorithms to remain fairly simple.

5.2. Optimized Update

The algorithms we have outlined above are efficient in terms of the number of attributes that they recompute when changes are made. However, they are not necessarily efficient in terms of the number of disk accesses needed. In this section we look at optimizations that Cactis uses to improve the number of disk accesses performed.

If we examine the Mark_Out_Of_Date and Eval_Attr routines which are central to the evaluation algorithm, we see that they are each just a traversal of part of the attribute dependency graph. In the naive algorithm, these traversals are implemented using straightforward recursive procedure calls, and hence represent a depth first ordering. In

the actual Cactis implementation, these traversals are performed explicitly by the system in an order determined at run-time and can therefore be optimized. Because all attribute evaluation rules are applicative in nature, we may in fact choose any traversal order which visits the same attributes. In particular, we are free to choose an order which reduces the number of disk accesses required.

In Cactis, we use an order of traversal which is chosen dynamically. The way we choose this order is to use a concurrent system in which a number of sub-traversals are (conceptually) running at the same time. Each time we reach a node which has two or more descendents to traverse, we fork a sub-traversal process to traverse the graph in each direction. For example, when we mark an attribute out-of-date, we then schedule a traversal process for each of the attributes which depend on it. When we evaluate an attribute, we request all the values needed to recompute its value in parallel. We can think of this as a parallel traversal of the graph where each branch of the traversal runs independently. To optimize disk access we use a greedy technique. Of all the sub-traversal processes which are runnable at any given time we will choose to execute the one which we expect to perform the least number of disk accesses.

In practice we will not create actual separate processes to accomplish our parallel traversal but instead simulate multiple processes in a single process. We break all computations into pieces or *chunks* to be scheduled independently. A chunk is a small piece of code that runs to completion and performs one small task. For example, a normal attribute evaluation rule is implemented using two chunks. The first schedules an evaluation for each of the other attribute values it depends on, then makes arrangements to schedule the second chunk when all the values are available. The second chunk, which is

scheduled only after all the values it needs have been computed, executes the attribute evaluation rule in order to compute the final value for the attribute. It then stores the value and informs any process waiting for the value that it is now available.

Processes in the system are each represented by what we call a *pending record.* A pending record is simply a data structure representing a pending computation. It contains bookkeeping information such as the name of the attribute involved, the number of values being waited for, storage for the values, an optional pointer to a list of *parent* pending records to be informed upon completion, and a pointer to a *chunk* routine to execute when all values are available. All chunk routines are constructed by the data language compiler discussed in section 6. The system works by removing a pending record from a priority queue of all currently runnable processes, and simply calling the chunk routine found in the pending record. This routine performs an appropriate action and terminates, at which time the system chooses a new process to run. The chunk routine for a process can perform actions such as computing a new attribute value or scheduling one or more new pending records. In addition, the chunk routine can, if needed, inform parent processes of completion. Informing a parent involves decrementing the number of values that the parent is waiting for (stored in the pending record for the parent) and, if this number falls to zero, moving the parent's pending record to the runnable queue.

The scheme for implementing simulated concurrency that we have described is very simple, easy to implement, and has proven quite efficient. The technique we use is similar to that used in the OWL real-time concurrent programming language. For additional information about implementation details, expected performance, translation of programs

into chunks and experience with the OWL language see [14].

Once concurrency has been introduced, the process of choosing a good traversal order simplifies to a scheduling problem. We choose a process to run which we expect to perform the least disk accesses. The obvious choice for this process is one which can be processed using attributes currently in memory. Note that each process is associated with one attribute, the one it is computing or marking out-of-date. It may need other attribute values to compute its own value, but these are the responsibility of other processes. Any needed values will have been collected in storage attached to the process' pending record before it is scheduled as runnable.

We use a simple hashing scheme to index all pending records by the objects that contain the attribute that they are associated with. Whenever a disk block is read into memory, all processes which are associated with some object stored on that block are promoted to a special very high priority queue. When new pending requests are scheduled, we first check to see if the object associated with the request is already in

---

Repeat
  Choose the most referenced object in the database that has not yet been assigned a block.
  Place this object in a new block.
  Repeat
    Choose the relationship belonging to some object assigned to the block such that:
      (1) The relationship is connected to an unassigned object outside the block and,
      (2) The total usage count for the relationship is the highest.
    Assign the object attached to this relationship to the block.
  Until the block is full.
Until all objects are assigned blocks.

Figure 6. Clustering Algorithm

memory, if so we schedule the request on the high priority queue. Since they can be executed without additional disk access, processes on the high priority queue always have priority over other processes.

In order to improve the locality of data references, we cluster data in the Cactis model on the basis of usage patterns. We keep a count of the total number of times each object in the database is accessed, as well as the number of times we cross a relationship between objects in the process of attribute evaluation or marking out-of-date. We then periodically reorganize the database on the basis of this information. In particular we pack the database into blocks using the greedy algorithm shown in Figure 6. This algorithm attempts to place objects which are frequently referenced together, in the same block. This tightens the locality of reference for the database and results in increased performance in databases where query streams exhibit commonalities over time. In fact, performance tests discussed in Section 7 indicate that proper clustering can result in savings of up to 60%. In addition to the clustering we have described, offline reorganization includes housekeeping tasks such as garbage detection and collection and the generation of statistics we use for scheduling, as described below.

Once all *in memory* processes have been executed we must choose a process to execute which will cause a disk access. We would like to choose the process which will cause the least disk accesses, however, we cannot know in advance which process this will be. What we will do instead, is use past behavior, or in the case of marking out-of-date a worst case estimate, as a predictor of future behavior. We keep information about past behavior in the form of a decaying average which changes over time. This makes the database self-adaptive, allowing changes in the structure of the database to be reflected in

changing averages and hence changing scheduling priorities.

In the Cactis data model, values flow across relationships in order to communicate information from one object to another. In order to provide statistics for self-adaptive optimization of the attribute evaluation process, we tag each relationship with a series of decaying averages. These statistics represent the average number of objects visited (or alternately the actual amount of disk I/O incurred) when each value transmitted across the relationship was requested in the past. We use these tags to assign a priority to pending records which are requesting values from across a relationship. The highest priority is given to the process with lowest expected disk I/O. Processes which request values local to an object rather than across a relationship are not of concern since they will be scheduled as high priority when the object is brought into memory. A special priority is given to processes which are the direct user requests that start a chain of computations.

In the case of the traversal which performs evaluation of an attribute, we update statistics when we return to the attribute in order to store its new value. However, in the case of the mark out-of-date traversal, we do not return to the object and hence cannot store an updated statistic. In this case we use an alternate worst case statistic computed when clustering was last performed. This statistic tells how many disk blocks will be visited in the worst case (i.e., assuming that no attributes to be visited are already marked out-of-date). A similar worst case statistic is used as an initial estimate for the dynamically changing decaying averages.

To summarize our strategy for optimized update, we treat the traversals needed to implement attribute evaluation as a concurrent computation. This allows us to dynamically choose a traversal order that reduces disk access. In this framework, the choice of a

traversal order simplifies to the choice of a scheduling order. Sub-traversal processes which can be executed without disk access are given highest scheduling priority. Once all computations that can be performed on in-memory data have been completed we choose processes which have the smallest expected number of disk accesses to run first. Expected disk accesses are measured by either using self-adaptive past performance statistics in the form of a decaying average, or on the basis of worst case statistics gathered at cluster time.

5.3. Triggers

While the Cactis data model is powerful, it is not as general as unconstrained trigger mechanisms. In particular, it is not possible to use the attribute evaluation strategy we have discussed to directly make structural changes to data. Because of the optimized implementation of derived data we can often, with a little thought, do without structural changes that would be needed under other models. However, in the cases where structural changes are required we must use a scheme equivalent to triggers. This scheme is compatible with the rest of the attribute evaluation mechanism, but can result in the same performance problems and unpredictable effects as normal triggers.

To implement triggers within the Cactis model, we have simply introduced attribute evaluation rules with side effects. In particular, the following evaluation rules will implement a trigger that should fire when a certain predicate becomes true.

```
P := pred(...);
LastP := Trigger(P,action_proc);
```

Here the predicate controlling the trigger is represented by the function pred and the routine which implements the action of the trigger is represented by action_proc. Function-

ally, the Trigger routine simply returns the value of P. However, it also "cheats" by looking at the current value of LastP. If the value of LastP is about to change from false to true Trigger() has the additional side effect of invoking action_proc. This technique can also be extended by including a retraction action to be executed when LastP is about to change from true to false. While the effects of the action_proc can cause an inefficient chain of other calculations involving triggers, we can at least use the efficient attribute evaluation algorithm to decide when to fire particular triggers.

5.4. Concurrency Control

Cactis uses a timestamping concurrency control technique [43]. Because of the possibility of an update involving a long chain of computations which touches many objects, a locking mechanism was judged too costly.

Concurrency control in Cactis is maintained at the level of individual attributes. This allows a significant amount of concurrency, but does involve the potentially significant space overhead. As Cactis is intended for applications with large, complex objects, the space required for timestamps may not prove to be that significant. As discussed in Section 5, each attribute has associated with it a read timestamp (**readtime**) and a write timestamp (**changetime**). To support rollback, a log mechanism was implemented. Standard timestamp logic is used; when a read or write conflict occurs, a transaction's log is deleted and the transaction is restarted with a new timestamp.

While it would seem that the transitive attribute dependencies that occur in a Cactis database would require more than local timestamp checks, this is not the case. Instead, all the required logic to properly handle transitive dependencies can be implemented as a part of the normal marking and evaluation phases of the attribute evaluation algorithm.

If a value is transitively needed to compute another value, its timestamps will be checked as a part of the (recursive) evaluation phase. The only unusual aspect of the system is that a request to read a value may cause both the read and write timestamps to be updated, since the value may need to be recomputed.

In this section we have looked at a number of the implementation details of the Cactis data model and seen how concurrent, self-adaptive techniques are used to optimize update. In the next section, we examine the Cactis data language.

6. The Cactis Data Language

Part of the Cactis implementation effort has been the construction of an object oriented data definition language (DDL) for the data model. This section will give an overview of the parts of this language which deal with maintaining functionally derived data. Details of the language constructs supporting the Sembase sub-system are described in [15, 22].

6.1. Notation and Structure

In the Cactis data model, and hence the Cactis DDL, information is structured as objects. These objects are organized into a type hierarchy using a multiple inheritance type system. Individual objects encapsulate a series of named and typed attribute values, along with attribute evaluation rules which define how these values can be derived. In addition, objects may be related to other objects by means of typed and directed relationships. Information about the object may be exported along these relationships. Consequently, the primary interface to an object is the set of values that it transmits along its relationships, and the set of values it expects to be transmitted to it, along those relation-

ships. Although relationships are directed to represent semantic concepts, derived data can flow in both directions across the relationship.

Since relationships in the Cactis data model are directed, we call each end of a relationship either a *black* or a *white connector*. In the Cactis DDL, relationship types are declared by stating the number, type, and direction of each of the values that flow across a relationship. Values may flow either from black to white, or from white to black. Once a relationship type has been declared, an object type can declare that it possesses a black or white connector of that relationship type. Two objects can be related if and only if one possess a black connector for a given relationship type and the other possess a white connector.

As an example of the use of the Cactis DDL, we will describe the declaration of objects which represent milestones in a software project. Milestones are a good example of the kind of highly interrelated data that the Cactis system is designed to handle. A milestone represents the scheduled and expected completion times for a single piece of work to be performed in the project. However, this piece of work may depend on the timely completion of other pieces of the project. Consequently, changing the expected completion time of a single milestone (i.e. slipping a deadline) can have important effects on the overall project. We cannot simply change a single milestone without checking how this affects other milestones. It is crucial that the complete effect of such a change be derived without omission or error, and that new milestones added to the system later can also automatically take such changes into account.

Figure 7, contains an example of the syntax used to declare the relationships and object types for milestones (we have modified the syntax slightly here to aid exposition).

```
Relationship Type milestone_dep Multi White
  Transmits
    exp_compl : time To Black; /* expected completion time comming into milestone */
End;


Relationship Type milestone_need Multi Black
  Transmits
    exp_compl : time To Black; /* expected completion time going out of milestone */
End;


/* Object to simulate many-many relationship for milestone dependencies */
Object Type mstone_many
  Relationships
    from_obj : Black milestone_need;
    to_obj   : White milestone_dep;
  Rules
    to_obj.exp_compl := from_obj.exp_compl;
End;


Object Type milestone
  Relationships
    depends_on      : Black milestone_dep; /* things this milestone depends on   */
    needed_by       : White milestone_need; /* things depending on this milestone */
  Attributes
    sched_compl             : time;  /* originally scheduled completion time */
    local_work              : time;  /* time to complete milestone alone */
    expected_completion     : time;
  Rules
    expected_completion
      := /* sum of local work and latest out of things depended on */
        local_work +
        Iterator latest : time
          Init 0
          For Each dep In depends_on Do
            latest := later_of(latest, dep.exp_compl);
        End;
    needed_by.exp_compl := expected_completion;
End;
```

Figure 7. Milestone Object Types

Figure 8. Milestone Objects

Figure 8, shows a graphical representation of several example objects of the types defined in Figure 7 illustrating how they would be connected in practice. In Figure 7, we first declare two relationship types milestone_dep and milestone_need. These relationships taken together are used to represent the many to many relationship of one milestone being dependent on another. Cactis relationships can be one to many, but not many to many. To implement many to many relationships between milestone objects we use an extra object of type mstone_many. This approach is similar to the strategy used to implement the set construct in the CODASYL model. As we have declared them, each relationship is one to many. We use objects of type mstone_many to connect this pair of one to many relationships into a many to many relationship. Note that these relationships transmit one value: exp_compl. This is the expected completion time of the milestone being depended upon.

Each milestone in the system is responsible for computing its own expected completion time based on the expected completion time of the things it depends on, along with internal information about how much work is required locally. The milestone object

```
Object Type monitored_milestone Subtype of milestone

  Attributes
    late : Important Boolean
  Rules
    late := later_than(expected_completion, sched_compl);

End;
```

Figure 9. Monitored Milestone

type shown in Figure 7 contains attributes and attribute evaluation rules to accomplish this. The attributes sched_compl and local_work are intrinsic attributes representing the originally scheduled completion time for the milestone and the amount of time needed to complete the milestone, respectively. The attribute expected_completion computes the actual expected completion time for the item. This is done using the Iterator facility of the Cactis DDL language to compute the latest of all the milestones that the current milestone depends on. This latest time is then added to the amount of local work to be done to obtain an expected completion time for the milestone (we have slightly simplified the computation here by not properly accounting for milestones which are not dependent on other milestones). The single attribute evaluation rule shown here can be used to keep the expected completion times for all milestones in an arbitrarily large project up to date automatically.

Once we have created an object type such as the milestone type shown in Figure 7, we can use the inheritance facility of the language to create a new object type: monitored_milestone, which adds a new attribute as shown in Figure 9. This attribute explicitly computes whether a milestone is expected to be late. This late attribute could

be used to alert the user of potential unexpected problems when changes are made to the database. Since this new object type offers a black milestone_dep connector and a white milestone_need connector, it can be substituted for any existing milestone object without disturbing the existing functionality of the system. In this case, a subtype of an existing object type was used. However, in general, objects of any type with the proper connectors could be substituted. Further, these new objects could be integrated with existing milestone objects without forcing them to be replaced. This is an example of how the Cactis data model is well suited to extending the functionality of an existing database while still supporting existing objects and functionality. This is particularly important for application domains such as software environments where we expect to add new objects and tools to the system during its use.

In addition to the features we have illustrated above, the Cactis DDL also offers the ability to create new attribute types using records, arrays, and a set of primitive types such as stings, characters, integers, booleans, and real numbers. Attribute evaluation rules are constructed using expressions built from a subset of the operators of the C language extended with the iterator shown above, as well as constructs for computing array and record valued expressions. Name equivalence typing is used throughout. Finally, the system provides the capability to invoke user supplied functions written in C, Pascal, or Fortran so that complex or expensive computations can be handled in a conventional programming language.

6.2. Implementation

The Cactis DDL translator is implemented in C using the lex and yacc compiler generation tools. It creates standard C code and data structures which are suitable for

input to the Unix C compiler.

Most of the implementation of the translator uses straightforward compiler techniques. The only unusual aspect of the translator is the way it treats expressions. As discussed in the last section, the optimized update algorithm used requires that expressions be broken into "chunks" which can be scheduled independently. The translator is responsible for breaking all expressions into these chunks. This is done using a general strategy where the values needed to compute an expression are all requested in parallel, then when all values are available, the expression itself is computed. This involves creating a chunk for requesting each value, and a chunk for evaluating the expression itself. In the case of conditional expressions and iterators, the expression evaluation chunk is further recursively broken into request and evaluation chunks. In all cases, the translator compiles request and evaluation chunks in such a way that the proper chunk is placed on the scheduling queue by other chunks at the proper time. This compile time analysis allows the run-time scheduler to be very simple and efficient.

7. Performance Tests

In this section we discuss a number of performance tests that have been run on Cactis. The purpose of these tests was not to benchmark Cactis against any known standard. Rather, we wanted to illustrate the effectiveness of two things: the priority mechanism of the scheduler and the algorithm used to cluster databases off-line. It is important to note that the self-adaptive nature of Cactis assumes that there is a certain amount of similarity of processing requests over time so that information about the past is in some way predictive of the future. However, even with little similarity in query history, the scheduler would still be able to take note of the blocks currently in memory and

give scheduling priority to processes that needed these blocks. Also, the clusterer would still provide some benefits over randomly placed data.

To perform the tests, we constructed test databases and query streams. In order to simulate repeated query sessions on one database, identical query streams were used. We realize that this is somewhat unrealistic and produces results that are better than they should be. But it was not possible to get any real feel for how much similarity would exist from one session to another in an actual Cactis application. In order to facilitate the construction of test databases, a database generator was constructed. It is described in the next subsection, then following that, the test results are summarized.

7.1. The Database Generator

The database generator creates random schema, data, and query streams tailored to specific parameters. In particular, the database generator can be used to create random databases in which aspects such as interconnection patterns, overall interconnection level, and number of relationship cycles in the data, can be varied under user control. This has allowed us to test the Cactis system on a range of different types of data.

The database generator accepts six pieces of information from the user to describe the characteristics of a generated database. This information includes:

Total_Size -

The total number of objects in the resulting database.

Connected_Size -

The size, in terms of number of objects, of each connected component of the resulting database.

Density -

A factor which determines the overall density of attributes within objects.

Cycle_Bias -

A bias factor which determines the likelyhood of relationship cycles occurring in the data.

Important_Bias -

A bias factor which determines the percentage of attributes in the data which will be designated important.

Templates -

A set of attribute dependency templates which determine the actual structure of attribute dependencies in the resulting database.

```
connected := empty;
unconnected := set of objects to be connected;
O1 := random object removed from unconnected;
Add O1 to connected;

Repeat
   O1 := random object in connected;
   If random() > Cycle_Bias Then
        /* no cycle created */
        O2 := random object removed from unconnected;
        Add O2 to connected;
   Else
        /* cycle created */
        O2 := random object in connected;
   Create relationship between O1 and O2;
Until unconnected is empty;
```

Figure 10. Random Connection Algorithm

```
Repeat Density times
  P := root node of random template tree chosen from Templates;
  O := random object chosen from current connected component;
  Root_attr := Layout(P,O);
  If random() < Important_Bias Then designate Root_attr important;

Layout(P,O):
  Place new attribute A in object O;
  For Each Child: P_child of P Do
     O2 := randomly selected object related to O;
     B := Layout(P_child, O2);
     Make attribute A dependent on attribute B;
  End;
  Create attribute evaluation rule for A which reflects dependencies created;
  Return(A);
```

Figure 11. Random Attribute Layout Algorithm

The generator works in two phases. In the first phase, it creates data objects and connects them via relationships. In the second phase it creates attributes to be placed in these objects, creates attribute dependencies to relate these attributes, and generates attribute evaluation rules corresponding to these dependencies.

Generated databases are partitioned into connected components. The first phase works by first creating Total_Size/Connected_Size objects to be placed in each connected component of the resulting database. The objects in each such component are connected via relationships using the algorithm shown in Figure 10. This algorithm establishes random relationships between objects in the connected component. The proportion of these relationships that represent cycles in the (at this point undirected) relationship graph is determined by the Cycle_Bias parameter.

After the objects of the connected components are created and related, the database generator proceeds to place attributes in these objects. This is done using the templates provided by the user. These templates are trees which indicate how a set of randomly created attributes should be related to one another. The system uses these template trees in the recursive algorithm shown in Figure 11 to assign attributes and attribute dependencies to objects. By varying the nature of the templates used along with the Cycle_Bias parameter, large random databases can be generated with a wide range of connectivity characteristics.

In addition to the connection characteristics of a created database, the user can also specify the characteristics of the query streams generated for the database. In particular, the user can modify the percentage of reads and writes along with the length of query streams.

7.2. Test Results

In the performance tests, all databases were of size 100 objects. The size of each connected component varied from 4% to 30%. The density was set to a default medium value. The cycle bias was ranged from low to medium, to high. In all tests, the important bias was at 30%. The template was a simple binary tree.

Cactis is intended for use in engineering design applications. Thus, we assume that, compared to a traditional business database, a Cactis database would have fewer and larger objects, and would have a significant amount of derived data. For this reason, we adjusted the sizes of the objects and the processing buffers so that only a handful of objects would fit in memory at one time. Further, we tested Cactis with a wide variety of cycle and connectivity settings.

Figure 12.

The generated query streams consisted of 5% updates. Each test consisted of running the same query stream five times, then reclustering. Then the query stream was run five more times, and reclustering performed again. Finally, the query stream was run five more times. This pattern was repeated for all permutations of connectivity, cycles, and

for each of the following scheduling algorithms: our optimized priority algorithm, and first come first serve.

Figure 13.

Figures 12 and 13 illustrate the effectiveness of the Cactis scheduling algorithm. Figure 12 contains three superimposed graphs, one for each of the three cycle bias settings (marked by a square, a circle, and a diamond). The vertical axis represents the savings in I/O reads when the priority scheduler is used instead of a first come, first serve algorithm. The savings is expressed as a percentage of the I/O count derived from using the first come, first serve algorithm. Figure 13 is a similar graph, except it does not compare first come, first serve with the priority algorithm. Rather, it compares two sessions which both use priority scheduling, but with clustering being performed in between. These graphs obviously exhibit fluctuations, but do indicate clear trends.

These results indicate four general patterns. The first three conclusions refer to the databases with medium and high cycles. First, for very low connectivity, the Cactis scheduler provides only a small improvement. Indeed, Cactis would not do well in many traditional business environments, where there is very little derived data. Second, for connectivities in the 12% to 20% range, the Cactis scheduler provides a substantial savings. Third, for very high connectivities (where over 30% of the database is interconnected) Cactis does not perform well. This makes sense; with a large chunk of the database interconnected, it is impossible to keep in memory a working set of blocks which may be used to satisfy a number of requests.

The fourth conclusion is that for all databases with low cycles, the Cactis scheduler does not perform as well. Again, this makes sense for the same reason very low connectivities lead to bad performance. A low level of cycles means that if a page is currently in memory, it is less likely to be needed by another computation currently outstanding. Thus, the priority scheduler is not able to make use of as much locality of reference in

selecting the tasks to perform. However, in the case of low cycles, the clusterer will provide a substantial improvement in I/O cost (see immediately below).

Figure 13 indicates the performance of Cactis when off-line clustering is used between two database sessions. Cactis was run twice for each data set, and both times, priority scheduling was used. The vertical axis shows the percent savings in I/O hits as a result of clustering. The results may be summarized very simply. Reclustering the data leads to a very substantial savings in cost in all cases, except when connectivity is very, very low. The reason for this is clear; with little connectivity, there isn't an effective way to recluster.

8. Limitations, Directions, and Conclusions

The current Cactis system has a number of limitations or unimplemented features. The system currently has no facilities for rollback or recovery except for those used in concurrency control. The data model does support an efficient undo capability which allows transactions to be rolled back and/or reapplied, however, this facility only works in a single user environment and not in a multi-user concurrent environment. This facility uses the property of the data model that all indirect updates automatically performed by changing one or more attribute values can be just as automatically undone simply by restoring the old value (a similar property holds for structural changes). Consequently, the same mechanism used to derive data, can also "underive" that data with equal ease. This capability is very useful in an interactive design environment, where the user may wish to try out alternatives and explore their effects without permanently committing to them.

A philosophically similar capability with an entirely different implementation approach can be found in the hypothetical databases approach discussed in [39, 40, 44]. Systems based on this concept us a differential file mechanism to explore various versions of a relational database. Currently research includes work on extending the Cactis single user undo system to operate in a multi-user environment.

In addition to the lack of a rollback and recovery mechanism, the current system also does not yet support conventional set-oriented queries. Also, because compiled C functions are used to define attribute evaluations rules, (see Section 5) we have little flexibility in dynamically changing the schema. Finally, the system does not provide authorization and security facilities, although it is unclear that these are of major importance in the engineering environments Cactis is intended to support.

As we have stated, the Cactis database is intended to support applications which require a rich modeling capability, in particular, engineering design applications. In order to test the effectiveness of the Cactis database in managing such complex interrelated data, we have begun to explore its use in the support of application areas such as software environments [19]. In addition, we are also in the process of constructing a distributed version of Cactis, with this effort just getting under way. As modern software environments will most likely be used in distributed workstation applications, this facility is viewed as crucial. It will be necessary to allow different users at different machines to configure their own environments privately and share information. Cactis is well suited to this task, as it allows the end user to conveniently tailor a local database. Also, the concurrent implementation of Cactis is naturally suited to a parallel or distributed system. In this way, various sub-traversals may actually be running at the same time. Additional

work is now underway to support replicated data in a distributed environment. Finally, research is now underway to support extensions to the logical data model and physical data model to improve efficiency and expressiveness.

To conclude, we have introduced a powerful data model based on derived data. This model is accompanied by a very simple incremental update algorithm which lends itself to an optimized self-adaptive physical implementation based on the selective scheduling of concurrent subtasks.

Acknowledgements

The authors would like to thank the team that constructed the software described in this paper: Pam Drew, Shehab Gamalel-din, Janet Jacobs, Deacon Lancaster, Carla Mowers, Loraine Neuberger, Evan Patten, Don Ravenscroft, Tom Rebman, Kurt Rivard, Jerry Thomas, and Gary Vanderlinden. In particular, Pam Drew and Tom Rebman deserve much credit for constructing Cactis, and Jerry Thomas constructed the data language processor and the database generator. Pam Drew also ran all the performance tests.

References

1. B. Alpern, A. Carle, B. Rosen, P. Sweeney and K. Zadeck, Incremental Evaluation of Attributed Graphs, *IBM Research Report RC 13205*, October 1987.
2. M. P. Atkinson and K. G. Kulkarni, Experimenting with the Functional Data Model, *Technical Report on Persistent Programming, University of Edinburgh 5*(Semptember, 1983),.

3. D. S. Batory, J. R. Barnett, F. F. Garza, K. P. Smith, K. Tsukauda, B. C. Twichell and T. E. Wise, GENESIS: A Reconfigurable Database Management System, *To appear in IEEE Transactions on Software Engineering*, .

4. D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel, CommonLoops: Merging Lisp and Object-Oriented Programming, *Conference Proceedings of OOPSLA '86*, Portland, Oregon, Sept. 1986, 17-29.

5. A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems 3*(October 1981), 353-387.

6. O. P. Buneman and E. K. Clemons, Efficiently Monitoring Relational Databases, *ACM Trans. Database Systems 4*(Sept. 1979), 368-382.

7. P. Buneman, R. E. Frankel and R. Nikhil, An Implementation Technique for Database Query Languages, *ACM Transactions on Database Systems 7*(June 1982), 164-186.

8. M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, Object and File Management in the EXODUS Extensible Database System, *Proceedings of the Twelfth International Conference on Very Large Databases*, August, 1986, 91-100.

9. A. Chan, S. Danberg, S. Fox, W. Lin, A. Nori and D. Ries, Storage and Access Structures to Support a Semantic Data Model, *Proceedings of the Eight International Conference on Very Large Databases*, September 8-10, 1982.

10. P. P. Chen, The Entity-Relationship Model--Toward a Unified View of Data, *ACM Trans. on Database Systems 1*, 1 (1976), 9-36.

11. K. Cooper, K. Kennedy and L. Torczon, The Impact of Interprocedure Analysis and Optimization in the Rn Programming Environment, *ACM Transactions on Programming Languages and Systems*, October 1986, 491-523.

12. A. Demers, T. Reps and T. Teitelbaum, Incremental Evaluation for Attribute Grammars with Application to Syntax Directed Editors, *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, 105-116.

13. K. Dittrich and U. Dayal, *International Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986.

14. M. D. Donner, The Design of OWL: A Language for Walking, *SIGPLAN Notices 18*(June 1983), 158-165.

15. D. Farmer, R. King and D. Myers, The Semantic Database Constructor, *IEEE Transactions on Software Engineering SE-11*(July 1985), 583-590.

16. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

17. M. Hammer and D. McLeod, Database Description with SDM: A Semantic Database Model, *ACM Trans. on Database Systems* 6, 3 (1981), 351-386.

18. S. Hudson and R. King, CACTIS: A Database System for Specifying Functionally-Defined Data, *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 26-37.

19. S. E. Hudson and R. King, Object-oriented database support for software environments, *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, California, May, 1987.

20. R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *USC Technical Report Tech. Rep.-86-201*(March 1987), .

21. L. Kerschberg and J. E. S. Pacheco, A Functional Data Base Model, Technical Report, Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, Brazil, February, 1976.

22. R. King, Sembase: A Semantic DBMS, *Proceedings of 1st Int'l Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984, 151-171.

23. R. King and D. McLeod, Semantic Database Models, in *Database Design*, S. B. Yao (ed.), Prentice Hall, 1985.

24. D. E. Knuth, Semantics of Context-Free Languages, *Math. Systems Theory J. 2*(June 1968), 127-145.

25. D. E. Knuth, Semantics of Context-Free Languages: Correction, *Math. Systems Theory J. 5*(Mar. 1971), 95-96.

26. G. M. E. Lafue and R. G. Smith, Implementation Of A Semantic Integrity Manager With A Knowledge Representation System, *Proc. First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 24-27, 1984, 172-185.

27. D. Maier, J. Stein, A. Otis and A. Purdy, Development of an Object-Oriented DBMS, *Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications*, September 29-October 2, 1986, 472-482.

28. D. Maier and J. Stein,, Indexing in an Object-Oriented DBMS, *Proceedings of the First International Workshop on Object-Oriented Database*, Pacific Grove, California, September 23-26, 1986, 171-182.

29. F. Manola and U. Dayal, " PDM: An Object-Oriented Data Model", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September

23-26, 1986, 18-25.

30. M. Morgenstern, The Role of Constraints in Databases, Expert Systems, and Knowledge Representation, *Proc. First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 24-27, 1984, 207-223.

31. J. R. Nestor, Recreation and Evolution in a Programming Environment, *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 230.

32. K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli and C. Jacobi, *The PASCAL <P>Compiler: Implementation Notes*, Instituts fur Informatik - ETH, Zurich, 1976.

33. M. H. Penedo, Prototyping a Project Master Data Base for Software Engineering Environments, *Proceedings of the Second Symposium on Practical Software Environments*, December 1986.

34. T. Reps, Optimal-time Incremental Semantic Analysis for Syntax-directed Editors, *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1982, 169-176.

35. T. Reps, T. Teitelbaum and A. Demers, Incremental Context-Dependent Analysis for Language-Based Editors, *ACM Trans. Prog. Lang. and Systems 5*(July 1983), 449-477.

36. A. Shepherd and L. Kerschberg, Constraint Management in Expert Database Systems, *Proc. First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 24-27, 1984, 522-546.

37. D. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Trans. on Database Systems 6*, 1 (1981), 140-173.

38. A. H. Skarra and S. B. Zdonik, The Management of Changing Types in an Object-Oriented Database, *Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications*, September 29-October 2, 1986, 483-495.

39. M. Stonebraker and K. Keller, Embedding Expert Knowledge and Hypothetical Databases into a Database System, *Proc. ACM SIGMOD Conf.*, 1980.

40. M. Stonebraker, Hypothetical Databases as Views, *Proc. ACM SIGMOD Conf*, 1981.

41. M. Stonebraker and L. A. Rowe, The Design of Postgres, *Proceedings of International Conference on the Management of Data*, May, 1986, pages 340-355.

42. R. Taylor, Arcadia: A Software Development Environment Research Project, *University of California at Irvine, Dept. of Information and Computer Scinece, Technical Report*, April 1986.

43. J. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982.

44. J. Woodfill and M. Stonebraker, An Implementation of Hypothetical Relations, *Proc. of the International Conference on Very Large Data Bases*, 1983.

45. C. Zaroliagis, P. Soupos, S. Goutas and D. Christodoulakis, The GRASPIN DB - A Syntax Directed, Language Independent Software Engineering Database, *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 235-236.

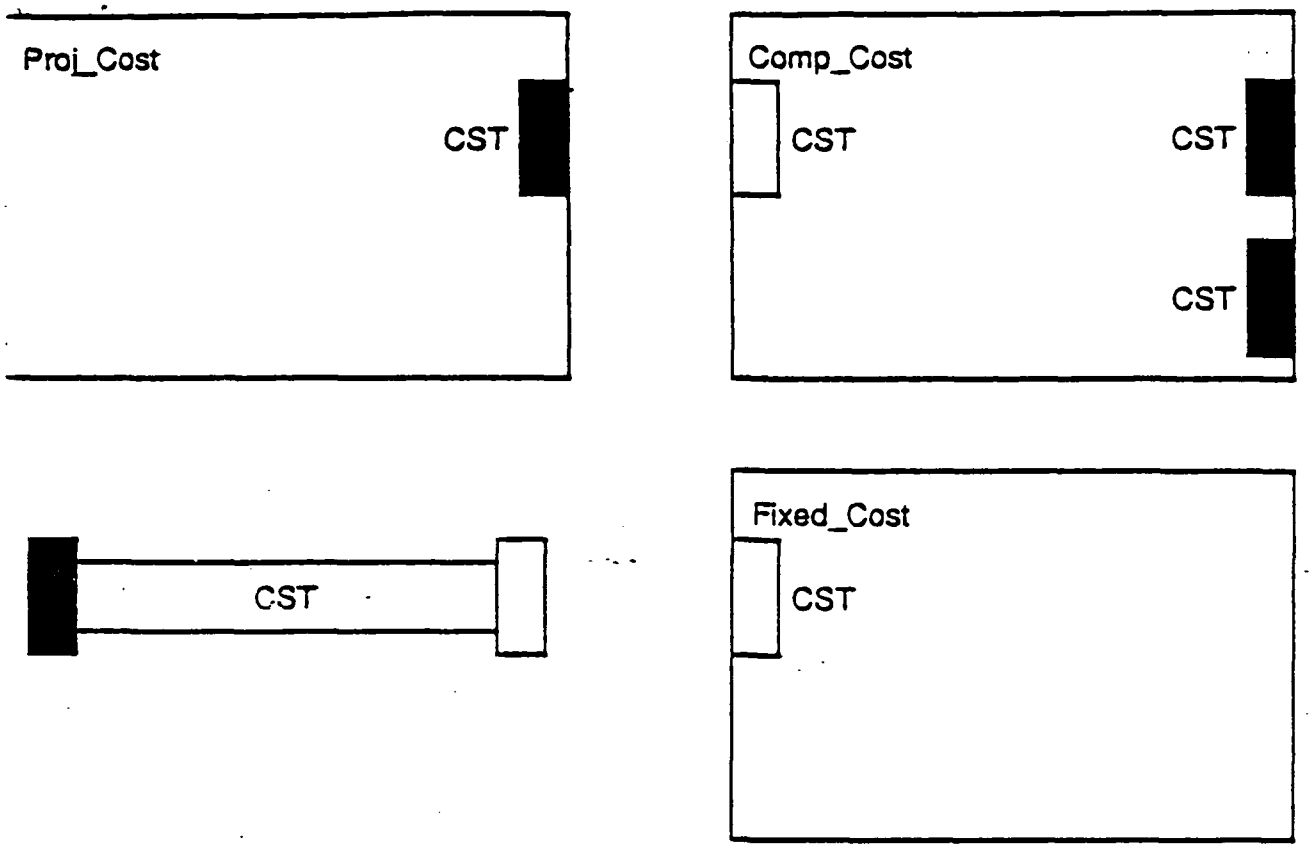


FIGURE 1

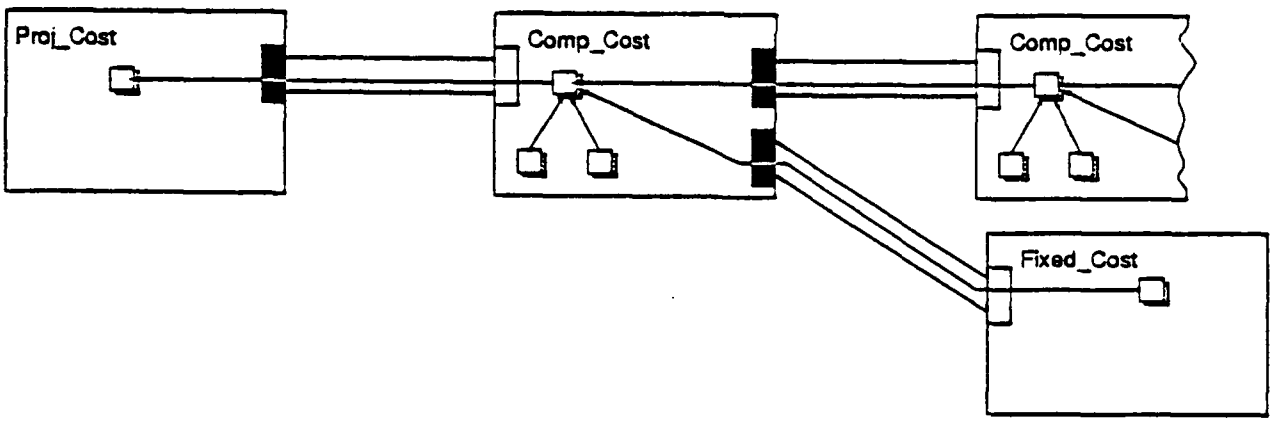
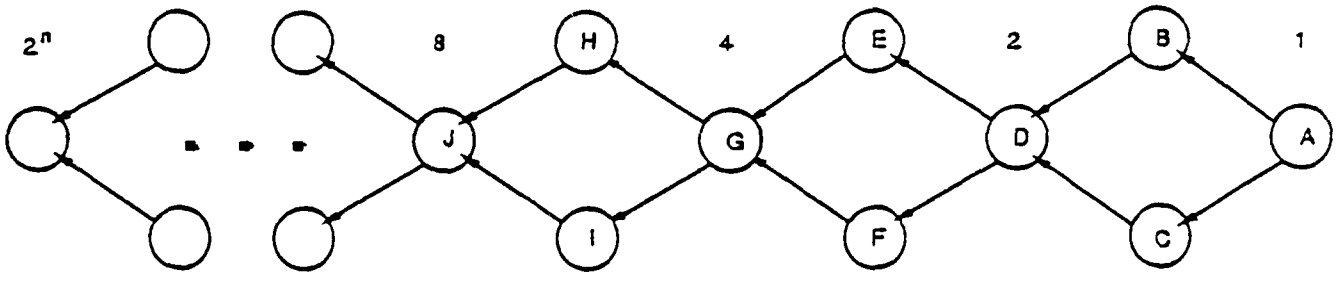


FIGURE —



FIGUREE 3

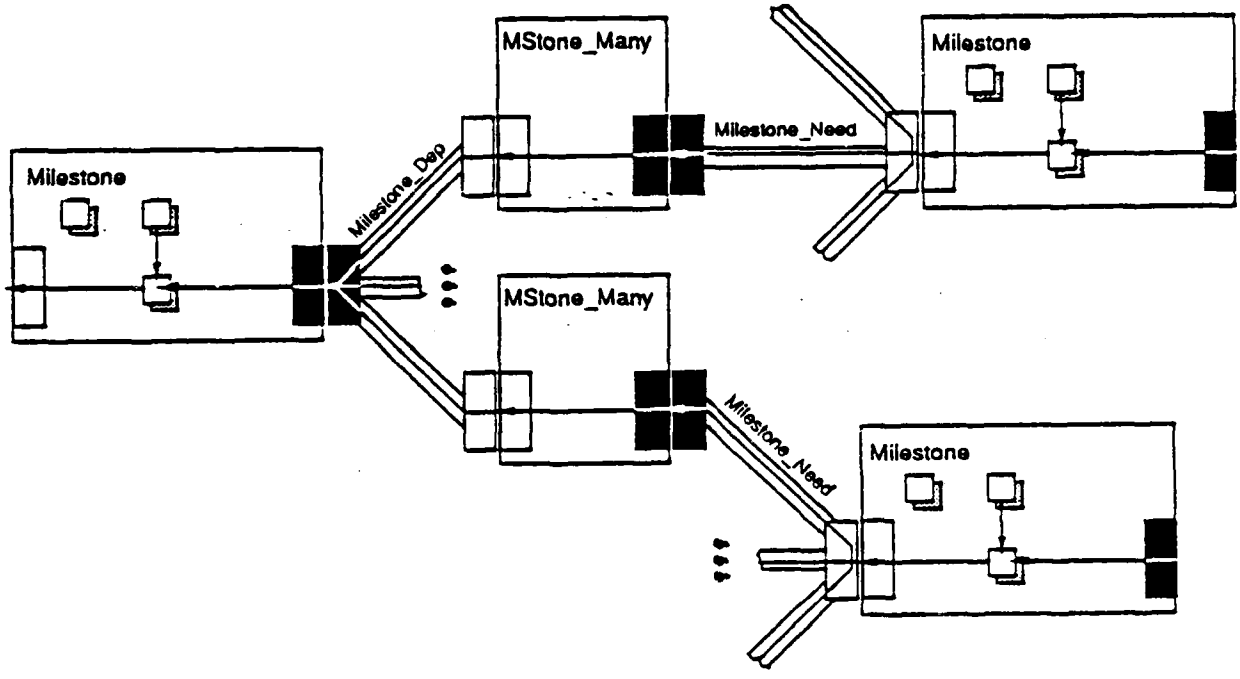
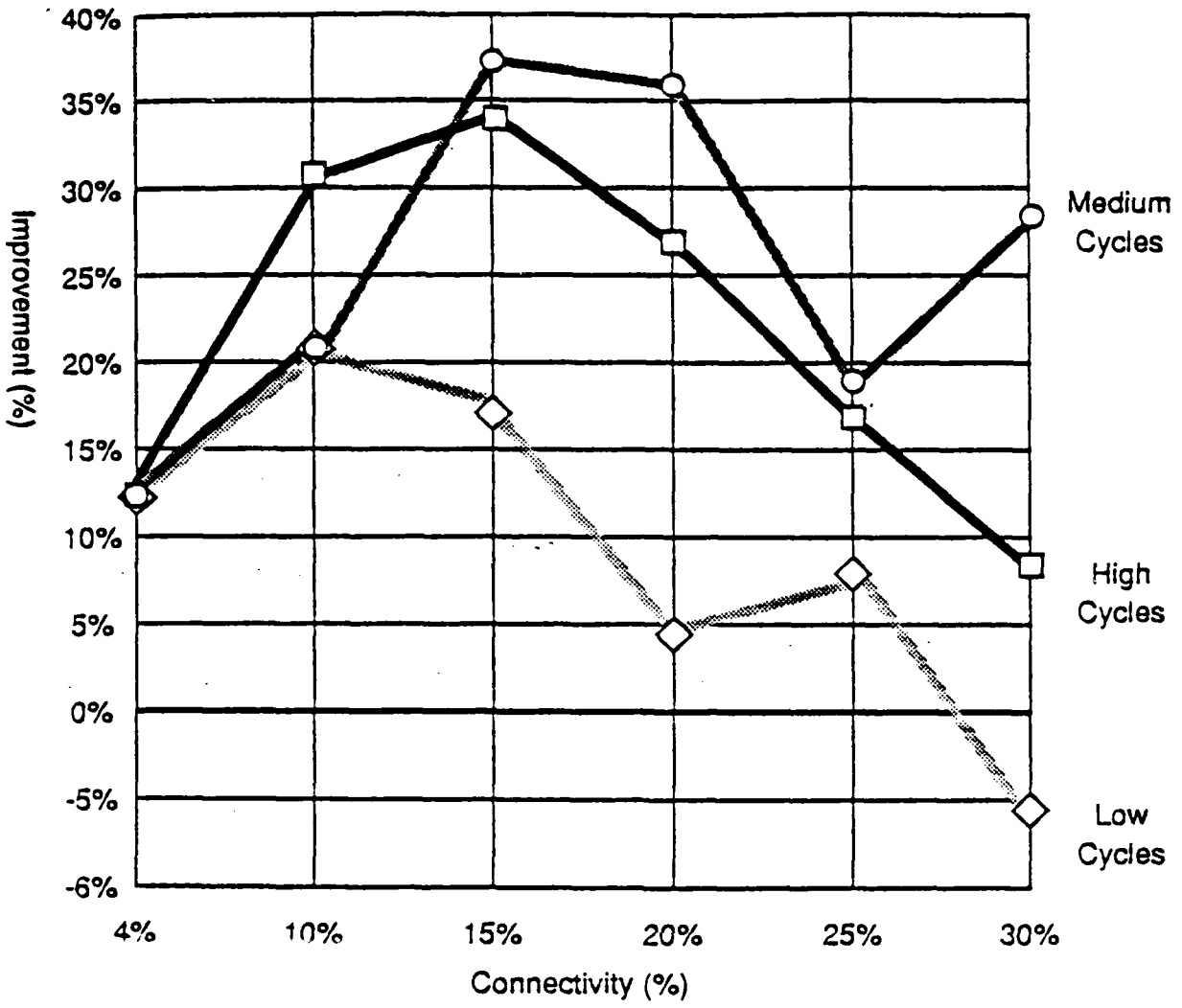
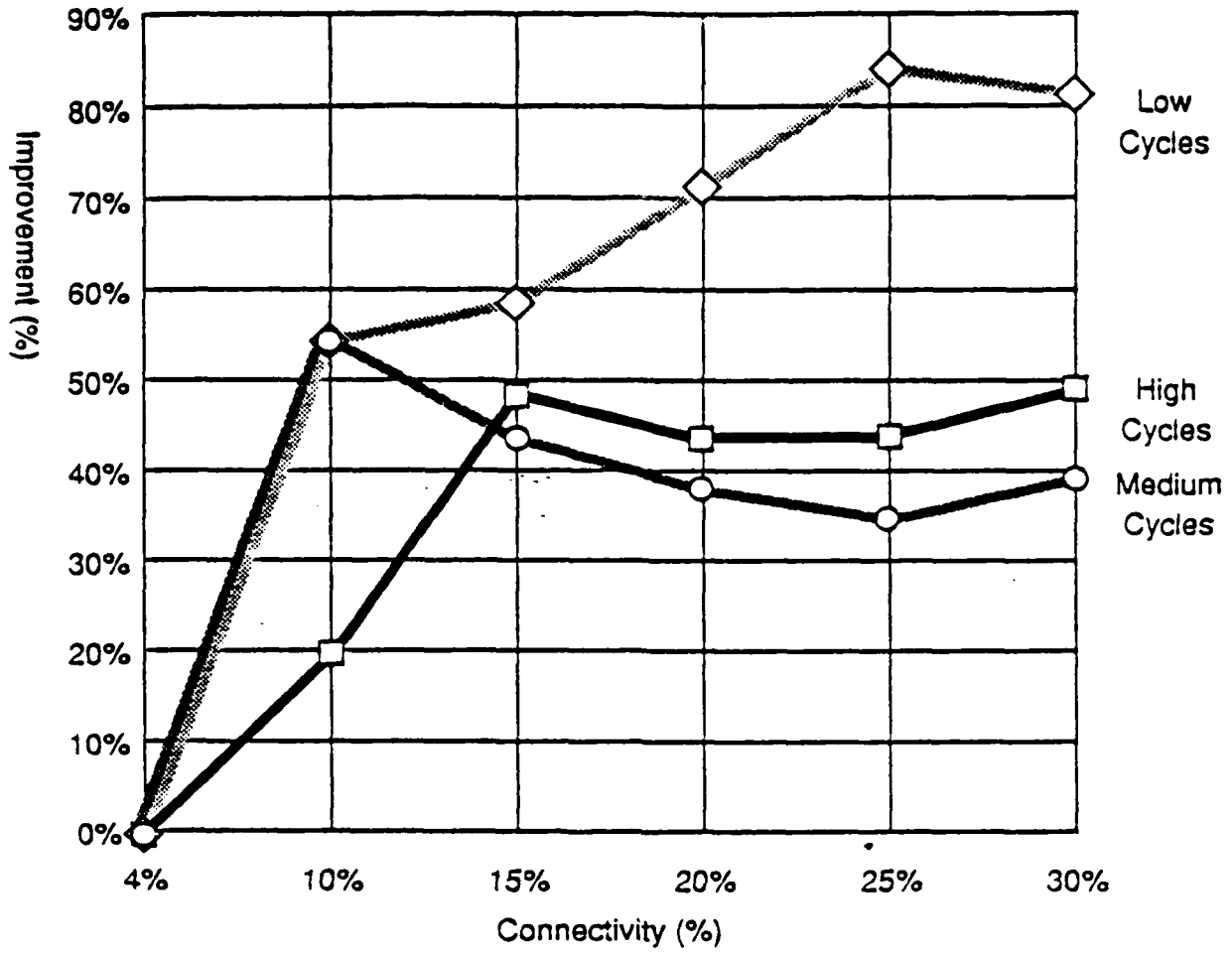

MStone_Many
Milestone
Milestone_Need
Milestone
Milestone_Dep
MStone_Many
Milestone_Need
Milestone

FIGURE 8



Improvement from Priority Scheduling
vs.
Connectivity

FIGURE 12



Improvement from Clustering
vs.
Connectivity

FIGURE 13

END

DATE

FILMED

DTIC

10-88